University of Zurich[UZH]

Department of Informatics

Federated SPARQL Query Processing

Reconciling Diversity, Flexibility and Performance on the Web of Data

Dissertation submitted to the Faculty of Economics,
Business Administration and Information Technology
of the University of Zurich

to obtain the degree of
Doktor der Wissenschaften, Dr. sc.
(corresponds to Doctor of Science, PhD)

presented by
Cosmin Adrian Bașca
from Romania

approved in July 2015

at the request of
Prof. Abraham Bernstein, Ph.D.
Prof. Stefan Schlobach, Ph.D.

University of Zurich[UZH]

The Faculty of Economics, Business Administration and Information Technology of the University of Zurich hereby authorizes the printing of this dissertation, without indicating an opinion of the views expressed in the work.

Zurich, July 15, 2015

Chairwoman of the Doctoral Board: Prof. Dr. Elaine M. Huang

To my family, for their love and support.
Pentru neamul meu.

Abstract

Querying the ever-growing Web of Data poses a significant challenge in today's Semantic Web. The complete lack of any centralised control leads to potentially arbitrary data distribution, high variability of latency between hosts participating in query answering, and, in the extreme, even the (sudden) unavailability of some hosts during query execution. In this thesis we address the question of how to efficiently query the Web of Data while taking into account its scale, diversity and unreliable and uncontrollable nature. We begin by first introducing Avalanche, a federated SPARQL engine which: *1*) makes no assumptions about RDF data distribution to SPARQL endpoints, *2*) is adaptive to changing network conditions, i.e, can adapt to slow network connections or endpoint unavailability, *3*) retrieves up-to-date results from SPARQL endpoints, and *4*) is flexible by making limiting assumptions about the structure of participating triple stores.

Tailored to address the *semantic heterogeneity* derived from the Web of Data's rich and broad semantic diversity, coupled with its characteristic lack of guarantees, Avalanche employs a *fragmented* query planning approach, under a concurrent and parallel execution model. By fragmented execution, we refer to the fact that the original SPARQL query is rewritten as the union of all *fragments* which comprise it. A *query fragment* is defined as the conjunction of all query triple patterns, where a triple pattern can be resolved by only one endpoint.

As the Web of Data continues to grow, we postulate that so is the likelihood that large numbers of endpoints will index data, sharing the same vocabularies, thus forming *semantically homogenous* partitions of the Semantic Web. Focusing on this scenario and in order to address some of Avalanche's limitations, we introduce x-Avalanche an extension of our original system. Here, we add support for disjunctions by using a distributed union operator capable of scaling to hundreds or thousands of endpoints. Furthermore, we enhance the distributed state management with: *a*) remote caches aimed to reduce the high latency typical of SPARQL endpoints, *b*) multicast parallel bind-joins exploiting the SPARQL 1.1 VALUES clause, and *c*) proxy based execution of x-Avalanche operators.

Finally, in x-Avalanche, we introduce a novel and parallel-friendly optimisation paradigm designed not only to offer an optimal tradeoff between total query execution time and fast first results, but also to consider an extended planning space unexplored so far, thus taking the *fragmented* execution model first introduced in Avalanche to its logical conclusion. Combined, x-Avalanche's enhancements and optimisations can lead to dramatic performance improvements over top performing state of the art federated SPARQL engines. To conclude, our results show that on average x-Avalanche can be more than one order of magnitude faster when executing SPARQL queries.

Zusammenfassung

Query-Abfragen im stetig wachsenden Web of Data stellen eine entscheidende Herausforderung für das Semantic Web dar. Das komplette fehlen jeglicher zentralen Kontrolle führt zu einer potentiell willkürlichen Verteilung von Daten, grossen Schwankungen bezüglich der Latenz von Hosts welche Queries beantworten und im Extremfall zum plötzlichen Ausfällen von einzelnen Hosts während der Query-Ausführung. In dieser Disertation behandeln wir die Frage nach der effizienten Verarbeitung von Queries unter Berücksichtigung der Grösse, Unzuverlässigkeit und unkontrollierbaren Natur des Web of Data. Als erstes führen wir Avalanche ein, eine Federated SPARQL Engine welche: *1*) keinerlei Annahmen bezüglich der Verteilung der RDF Daten macht, *2*) sich an sich ändernde Netzwerkbedingungen, d.h. langsame Verbindungen oder Nichtverfügbarkeit von Endpoints, anpasst, *3*) aktuelle Resultate von SPARQL Endpoints empfängt und *4*) flexibel ist dank nur schwachen Annahmen welche bezüglich der teilnehmenden Triple Stores gemacht werden. Um die grosse Diversität in der Semantik des Web of Data, welche zu einer Heterogenität derselben führt, gepaart mit dem fehlen jeglicher Zusicherungen, zu bewältigen, nutzt Avalanche einen fragmentierten Ansatz zur Query Planung unter einem nebenläufigem Ausführungsmodel. Unter einem fragmentierten Ansatz verstehen wir das umschreiben von SPARQL Queries als UND-Verknüpfung aller Fragmente die den Query einschliessen. Ein Query Fragment definieren wir als die UND-Verknüpfung aller Triple-Pattern, wobei ein Pattern jeweils nur von einem Endpoint bearbeitet werden kann.

Wir postulieren, dass mit dem Wachstum des Web of Data auch die Wahrscheinlichkeit wächst, dass eine Vielzahl von Endpoints Daten katalogisieren, welche das gleiche Vokabular teilen und so semantisch homogene Teile des Semantic Web formen. Um diesem Szenario gerecht zu werden und um uns an die Grenzen von Avalanche zu richten, führen wir x-Avalanche ein, welches eine Erweiterung des ursprünglichen Systems darstellt. In dieser Erweiterung führen wir die Unterstützung für ODER-Verknüpfungen ein indem wir einen verteilten Vereinigungs-Operator einführen, welcher bis hunderte oder tausende von Endpoints skaliert. Ausserdem verbessern wir die Verwaltung von verteilten Zuständen durch *a*) Remote-Caches um die hohe Latenz von typischen SPARQL Endpoints zu reduzieren, *b*) parallele Multicast Bind-Joins welche die VALUES-Klausel von SPARQL 1.1 ausnutzten und *c*) Proxy-basierte Ausführung von x-Avalanche Operatoren. Schliesslich führen wir in x-Avalanche einen neues Optimierungs-Paradigma ein welches die Parallelisierung erleichtert und entwickelt ist um nicht nur einen optimalen Kompromiss zwischen gesamter Laufzeit eines Queries und schnellen ersten Resultaten zu bieten, sondern auch um ein bis jetzt unergründetes Gebiet der Query Planung zu berücksichtigen, welches das fragmentierte Ausführungsmodel, eingeführt in Avalanche, zu dessen logischen Abschluss bringt. Die Verbesserungen und Optimierungen von x-Avalanche können gemeinsam zu drastischen Leistungsverbesserungen gegenüber anderen aktuellen Federated SPARQL Engines

führen. Abschliessend zeigen unsere Resultate, dass X-AVALANCHE im Schnitt mehr als eine Zehnerpotenz schneller sein kann bei der Ausführung von Queries.

Acknowledgements

First and foremost I would like to thank my supervisor Abraham (Avi) Bernstein for giving me the opportunity to pursue a PhD under his supervision at the University of Zürich. The support and research freedom granted to me made the work in this thesis possible. I greatly appreciate all his contributions of time, ideas, and funding to make my Ph.D. experience productive and stimulating. I would also like to thank my co-examiner Stefan Schlobach for making the time to co-referee this work and for his invaluable feedback.

My friends and colleagues at the Department of Computer Science (IFI), have contributed immensely to my personal and professional time at the University of Zürich. I would like to extend a *thank you* to all of them for their friendship, support and fun times together. In no particular order, thank you: Kathy, Cathrin, Jonas, Adrian, Amancio, Lorenz, Philip, Barbara, Bibek, Christian, Daniel (Spicar), Daniel (Strebel), Mihaela, Floarea, Helen, Hanspeter, Rico, Beat, Iris, Jayalath, Juk, Khoa, Marc, Mike (Feldman), Mike (Shann), Patrick (de Boer), Patrick (Minder), Shen, Timo, Tobias, Tom, and Rob.

A special *thank you* goes to my family. Words cannot express how grateful I am to my parents and sister for all their support and sacrifices made on my behalf. Your prayers were what sustained me thus far. Lastly, I would like to thank my love and best friend Ioana for her support, love and strength. Her help and smile has been indispensable during this time.

Financing

This work was partially supported by the Swiss National Science Foundation under contract number 200021-118000.

Table of Contents

Part I

Synopsis

Chapter 1

Introduction

For more than a decade the Semantic Web has slowly but consistently matured and evolved from the existing Web – a web of documents. First proposed in 2001, the Semantic Web [Berners-Lee et al., 2001] is, according to Tim Berners-Lee, *"a web of data that can be processed directly and indirectly by machines"*. It was designed to augment and enhance the existing Web with machine readable, semi-structured data. In the beginning, it exhibited a slow growth and adoption. The first metadata standards were proposed as early as the 1997 Meta Content Framework[1]. These soon developed into the RDF[2] and RDFS[3] W3C standards, which later became foundational to the Semantic Web. As stated in [Shadbolt et al., 2006], by 2006, the vision of the Semantic Web *"remains largely unrealised"*. However, a ripe and inter-disciplinary field, the Semantic Web soon started to attract more research leading to a globally rich Web of Data. Stemming from this original vision, one particular development of the Semantic Web is the Linked Data [Bizer et al., 2009b] movement, materialised as the Linked Open Data project[4], partly illustrated in Figure 1.

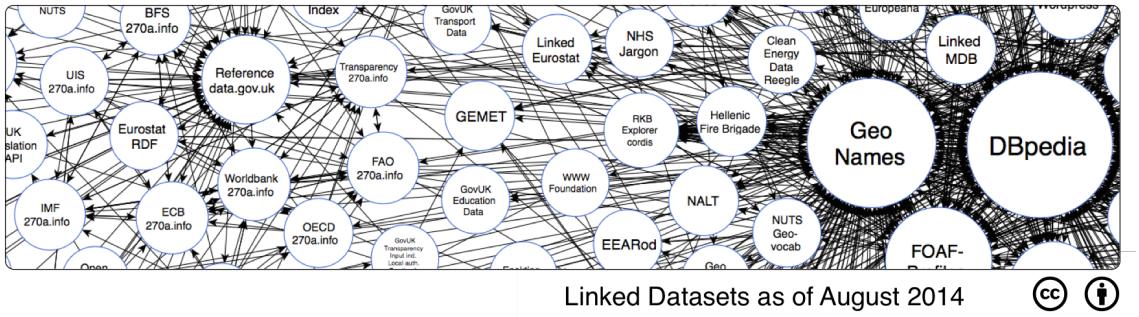


Fig. 1: Part of the Linking Open Data cloud, August 2014. Diagram by M. Schmachtenberg, C. Bizer, A. Jentzsch & R. Cyganiak. http://lod-cloud.net/.

Based on a simple set of guiding principles and standards, Linked Data's popularity started to grow at an unprecedented rate spawning both academic and industrial interest. It has grown from a handful of datasets in 2007 to more than 60 billion assertions about the real world, spread over more than 1000 datasets, as of August 2014 [Schmachtenberg et al., 2014].

[1] http://www.w3.org/TR/NOTE-MCF-XML/
[2] http://www.w3.org/TR/rdf11-primer/
[3] http://www.w3.org/TR/rdf-schema/
[4] http://linkeddata.org/

1 Motivation and Background

The dual nature of the Web of Data creates a fertile research ground by the multitude of problems that are endemic to it. On the one hand, the rapid adoption of data querying standards like SPARQL[5] and the SPARQL Protocol allow a data surfing agent to see the Web of Data as one logically centralised but physically distributed global database. On the other hand, just like on the Web, data is usually accessed via a single communications protocol, i.e., HTTP with encoded SPARQL queries, and uses a common representation format, i.e., RDF, which allows for links to be embedded. Consequently, on the Web of Data one cannot exercise the same level of control and management critical to ensuring a high quality of service (QoS), a usual characteristic of traditional distributed databases.

The traditional *optimise-then-execute* paradigm was the preferred query execution model since the introduction of System R [Astrahan et al., 1976]. This strategy yielded excellent results when there were few correlations between relations and there existed an abundance of statistical information about the indexed data. However, viewed as a globally distributed database, the Web of Data does not meet these characteristics. As identified in [Deshpande et al., 2006, Umbrich et al., 2013], some of the paramount factors are:

- **no guarantee of a functioning network**, on the Web, and hence the Web of Data, no guarantees can be made about latency, bandwidth and site or endpoint availability.
- **inaccurate statistics**, i.e., as data size continues to grow keeping track of accurate descriptive statistics becomes costly. This gives way to less accurate cost models, e.g., *selectivity estimation* based models [Başca and Bernstein, 2014, Görlitz and Staab, 2011, Stocker et al., 2008a] and rule-based or heuristic optimisers [Nikolov et al., 2013, Schwarte et al., 2011, Tsialiamanis et al., 2012]. Furthermore, relying on estimations when queries are complex only exacerbates the problem [Ioannidis and Christodoulakis, 1991], e.g., estimating join cardinality is notoriously difficult in the absence of advanced statistics.
- **dynamic data**, just like on the Web, Semantic Web data can change without any prior announcement.
- **dynamic workload** is a characteristic of the Semantic Web's rich domain diversity. For example, updates are less frequent for geographical information systems (GiS) data like *GeoNames*[6] than for domains like weather monitoring [Ghislain et al., 2013], yet on the Web of Data both data sources could contribute to the same query.
- **no control over data distribution and access policy**, is a fact attributed to the openness that is characteristic to the Web of Data. For example the datasets that constitute the Linked Open Data cloud, are published by different managing entities that can impose different copyright and intellectual property policies over (parts of) the data.

[5] http://www.w3.org/TR/sparql11-overview/
[6] http://www.geonames.org/

The above issues give rise to the main research question that led to the creation of this thesis:

Research Question: *How can the Web of Data be queried efficiently while preserving its flexibility, considering its scale, diversity, unreliable and uncontrollable nature?.*

A number of approaches that address this question have been proposed, ranging from centralised storage and indexing solutions to applying distributed query processing and traversal of the Linked Data ecosystem.

A conceptually simple solution would be to keep an up-to-date queryable endpoint of the entire Web of Data, achievable by storing all data in a *data warehouse*. This guarantees result-set completeness over the indexed data. In addition, since all data is under centralised control, high-performance traditional database optimisations can lead to efficient querying. However, such a solution breaks the flexibility of the Web of Data, by managing a copy of it. Furthermore, as argued in [Alstyne et al., 1995], data quality issues caused by incentive miss-alignments will inevitably lead to inefficiencies when treating the Web of Data as *one big datastore*.

Other solutions rely on the guiding principles of Linked Data, specifically: *1*) use URIs to identify *things*, *2*) use HTTP to dereference these URIs, *3*) describe these *things* when looked-up, making use of standards such as RDF and SPARQL and *4*) link to other *things* using their HTTP URI, when publishing new data. Following these four guidelines, systems and algorithms like the ones presented in [Hartig, 2013, Hartig et al., 2009a] resort to traversing the Linked Data while resolving the SPARQL query. The primary advantages of such an approach are the on-demand nature of the querying process, which leads to live access to data and therefore up-to-date results. However, they:

- cannot guarantee result-set completeness as the servers hosting the documents become unavailable,
- can suffer from performance penalties, i.e., overloaded servers that try to service "popular" URIs or servers behind high-latency connections, and
- are biased by the starting point of the search.

More recent solutions to querying the Web of Data rely on SPARQL endpoints, query- and update-able interfaces to the Web of Data typically backed by an RDF or triple store. While some RDF stores like OpenLink's Virtuoso[7] leverage decades of Relational Database Management Systems (RDBMS) development and optimisation by mapping the RDF data model to a relational schema, others rely on traditional Graph storage technology [Iordanov, 2010, Martínez-Bazan et al., 2007, Robinson et al., 2013] or develop RDF tailored native storage and indexing solutions [Neumann and Weikum, 2009b, 2010, Weiss et al., 2008]. These type of solutions form *Federated DBMSs*. The main advantages of federations of SPARQL endpoints are:

- result-set completeness if all pertinent endpoints are identified and no catastrophic failures occur during query processing,

[7] https://github.com/openlink/virtuoso-opensource

– up to date results from the indexed Web of Data – Linked Data documents not available via a SPARQL endpoint are not directly accessible,
– flexible integration of SPARQL endpoints, and
– efficient query processing, i.e., advanced database optimisation techniques can be applied.

Finally, some solutions build upon *peer-to-peer* (p2p) technology to efficiently and robustly provide answers to SPARQL queries [Cai and Frank, 2004, Nejdl et al., 2002]. However, these solutions break the flexibility of the Web of Data by assuming control over the distribution of data to sites, an unrealistic assumption on the Web of Data.

2 Research Hypotheses

Considering the complex nature of the problems endemic to the Web of Data, in the following we derive a set of hypotheses representing the foundation of this thesis.

Hypothesis 1: Retrieving triple-pattern cardinalities on-the-fly, at query time, can have a negligible impact on overall query performance while considerably simplifying the catalog of a federated RDF engine.

As mentioned in the previous section, a data federation solution has the potential to preserve both flexibility and performance when faced with the problem of querying the Web of Data. While traditional FDBMS's are built around a *catalog*, Hypothesis 1 assumes that it is possible to obtain similar levels of performance while querying, without relying on a harder to maintain catalog. The benefits of Hypothesis 1 are manyfold. First, the flexibility of the federation is increased, as joining the federation is not conditioned by the export of fine-grained statistics. Second, the federation is easier to maintain since the catalog contains coarse-grained information about participating endpoints, like location and the list of used vocabularies. Finally, by not requiring RDF stores to disclose fine-grained statistics, data ownership and privacy are preserved to a larger degree than in traditional cases. Furthermore, we assume that by aiding data privacy and ownership, the Web of Data benefits through the inclusion of datasets that fall under more restrictive copyright laws or access policies, like for example sensitive personal and medical data.

Hypothesis 2: Adaptive query execution improves federated SPARQL query processing time.

Web of Data federated SPARQL query processing can be affected to a large degree by the environment's uncontrollable nature. In consequence, adaptive query processing methods must be considered since no guarantees can be made about: *1*) the condition of the network, i.e., endpoint accessibility or connection latency being out of the data federation's control, and *2*) data distribution and stability. Hypothesis 2 assumes that adapting to changing environment conditions like network or data distribution also leads to an increase in performance when executing federated SPARQL queries. Furthermore, adaptivity also safeguards to some extent against the negative effects of relying on inaccurate descriptive statistics used during query planning and optimisation.

Hypothesis 3: Plan space fragmentation can offer a controllable and optimal tradeoff between time to first results and total SPARQL query execution time.

Often, applications need to make quick decisions or quickly render partial information to a user before all query results become available. Most SPARQL federated query processing systems either use a blocking execution design where all query results are returned at once, or make use of result-set *iterators*, allowing the caller to go through the result tuples one by one. This is often achieved by employing non-blocking physical join operator implementations like in [Acosta et al., 2011, Ladwig and Tran, 2011]. However, none of these approaches can exercise control over the optimal performance tradeoff between the time to obtain first results and total query execution time. In Hypothesis 3 we assume that by *fragmenting* the query planning space, such an optimal tradeoff can be achieved. Fragmentation is the process of partitioning the amount of work needed to resolve the query into a predefined number of groups or sub-queries. This gives the optimiser the freedom to *schedule* the execution of each fragment or sub-query given its cost. In addition, this is also beneficial to multi-query optimisation scenarios. Here the optimiser has the liberty to optimally intertwine the execution of different query fragments in order to maximise throughput and therefore quality of service.

Hypothesis 4: A stateful distributed SPARQL query protocol design, can dramatically improve the performance of federated SPARQL query processing.

The original W3C SPARQL specification did not provide support for executing federated queries. This fact was addressed with the adoption of the SPARQL 1.1 W3C recommendation[8], which included an extension supporting federated queries. Federating a SPARQL query came primarily through the use of:

- the SERVICE pattern, which allows the caller to specify the endpoint where the graph pattern can be executed, and
- the VALUES clause, which allows the caller to restrict the execution of a graph pattern to a set of given variable bindings.

However, by embracing a simple and robust design, the SPARQL protocol remains a stateless protocol. Hypothesis 4 assumes that federated SPARQL query processing performance can be improved by extending the stateless SPARQL protocol with distributed state management. It stands to reason that, by allowing a federation engine to manage and keep track of remote state on participating endpoints, the optimiser can make use of available resources more efficiently and minimise network communication when bottlenecks occur. This is not unlike peer-to-peer systems, that attain low latency responses and robustness to failures.

[8] http://www.w3.org/TR/sparql11-query/

Chapter 2

Contribution

Next we introduce the contributions of this thesis, while briefly evaluating them against the research hypotheses proposed earlier. In short the main contributions of this thesis are:

- Avalanche: a federated SPARQL processing system over the indexed Web of Data. Unlike traditional federated query engines, Avalanche does not require fine-grained prior knowledge about data distribution. In addition, in order to enhance performance, Avalanche employs both adaptive query processing methods and makes use of a stateful querying protocol based on the SPARQL 1.1 federation extension.
- *Query plan fragmentation*: a novel optimisation and parallel-friendly execution method, which features the optimal tradeoff between time to total and first results, given user or environment constraints.
- x-Avalanche: an extension of the Avalanche federated query engine. It features an enhanced distributed state management protocol based on scalable and efficient physical operator design for both unions and joins.

In the following we will detail each contribution and highlight how each hypothesis is supported by this thesis.

3 Avalanche

The main contribution of this thesis is materialised as the Avalanche SPARQL federated processing system. Avalanche is presented in detail in Chapter 5.

A number of solutions to querying the Web of Data have been proposed, ranging from centralised approaches to link traversal and federated query processing. One of the most flexible approaches, which has the potential for high performance optimisations typical of traditional distributed database management systems is that of a *data federation*. So far, much of the growth of the Web of Data has been subsidised through public funding, which led to the growth of the Linked Open Data cloud to more than 60 billion triples spread over more than 1000 datasets. However, many data sources are still not benefiting from the flexible integration and interlinking capabilities that the Web of Data can offer, primarily due to their sensitive nature. For example the Data Without Boundaries (DwB) project[9] is an effort of the European Union to facilitate *equal and easy access to official microdata* for researchers. The project deals with confidential microdata at the personal, household or institutional level. Such data is sensitive and subject to

[9] http://www.dwbproject.org/

legal restrictions requiring that data be *physically in the facilities of the data owners*. It is therefore critical in such cases that ownership is exercised at all times.

Traditional data federations require the publication of advanced statistics in a logically centralised component: the *catalog*, to aid during query planning and execution. This can represent a breach of the ownership assumption. In Avalanche, the catalog is not required to contain advanced statistical information, instead query pertinent fine-grained statistics are obtained during query execution, allowing for (parts of) sensitive data to be accessible during querying. Naturally, access policies need to be defined. A common concern that arises in this case is that obtaining such statistics, i.e., triple pattern cardinalities, can be a costly operation. However, this is not the case, since RDF stores like the ones in [Neumann and Weikum, 2009b, 2010, Weiss et al., 2008] offer fast and inexpensive access to triple pattern cardinalities. A secondary option would be to store voID[10] statistics that can be retrieved during query execution. Avalanche's performance, presented in Chapters 5 and 6, support Hypothesis 1.

In addition to quickly retrieving query pertinent statistics on-the-fly during querying, Avalanche also employs an adaptive execution pipeline where optimisation is intertwined with execution. This helps Avalanche mitigate to some extent issues like:

- source unavailability,
- high network connection latency, and
- unreliable estimates during query planning.

When benchmarked under changing network conditions, Avalanche's performance results support Hypothesis 2. Additionally, Avalanche utilises an early version of a custom stateful federated query protocol to increase query performance. The protocol relies on the SPARQL 1.1 federation extensions. Like peer-to-peer systems, Avalanche does not assume any centralised control, which allows for any participating endpoint to assume the role of the query-broker. Results presented in Chapter 5 also validate Hypothesis 4.

4 Query Plan Fragmentation

Query optimisation has been the primary driving factor behind the success of both free and commercial databases ever since their introduction. In general, optimisation algorithms are either *deterministic* or *randomised*. Deterministic optimisers can be further classified into *rule-based* or *heuristic*, and *cost-based* optimisers. While heuristic optimisers are used to handle large planning spaces, given their typical polynomial space and time complexities, they do not offer optimality guarantees. In contrast, cost-based optimisers are typically exhaustive in nature, and feature exponential time and space complexity but offer optimality guarantees – given the cost model. A popular exhaustive strategy is *Dynamic Programming* (DP) [Bellman, 1957]. In the distributed case, when data is partitioned to multiple sites traditional DP optimisers do not consider partition groupings during the logical plan construction phase, therefore avoiding the exploration of an *extended planning space* [Herodotou et al., 2011].

[10] http://www.w3.org/TR/void/

We propose *plan fragmentation* as a method to target this *extended planning space*. Fragmentation works by finding the optimal set of partition groups or endpoint disjunctions for each of the queries triple patterns. The result is a set of *fragmented bushy plans*, which are variants of bushy plans where the top subtrees represent disjoint fragments of the original query plan. One of their main advantage is that they can be parallelized easily and offer the optimiser control over scheduling the amount of work required by query execution. This, in turn, leads to achieving an optimal tradeoff between the time it takes to find first results and total query execution time. Results presented in Chapter 6 support Hypothesis 3.

5 x-Avalanche

In this thesis we distinguish between a query's result-set selectivity and its source selectivity, referring to the latter as *semantic selectivity*. While result-set selectivity refers to the cardinality of the query result-set, the source selectivity refers to the number of endpoints or sources that a query needs in order to produce answers. Queries that are highly source selective involve few sources (ideally one) during execution. Given the Web of Data's schema richness and broad semantic diversity, real-world and benchmark queries are typically *semantically selective*. Meaning that vocabularies bound to the query restrict its execution to only a handful of endpoints, considerably reducing the size of the problem. For example, the life science queries from FedBench [Schmidt et al., 2011] target only four sources of the entire benchmark. There are however, important situations for which this assumption does not hold:

- Large numbers of endpoints that store data using overlapping or the same set of vocabularies. This is not an unreasonable scenario as the Web of Data continues to grow.
- When SPARQL queries get rewritten to address endpoints which use similar yet overlapping vocabularies, in order to encompass more relevant data.

Hypothesis 4 is supported by results from Chapter 6 even in the more difficult setup of very large homogenous SPARQL federations, leading to improvements of more than one order of magnitude over the top performing state of the art federation engine [Saleem et al., 2014]. To achieve this, x-Avalanche enhanced its original federated SPARQL processing protocol with:

- *parallel multicast joins*, where each x-Avalanche endpoint can multicast and coordinate a join between several remote endpoints. To minimise network traffic x-Avalanche makes use ob bind-joins relying on the SPARQL 1.1 VALUES clause.
- *execution by proxy*, where all x-Avalanche operators can be executed directly or by proxy. Proxy based execution helps the query coordinator to offload orchestration effort to remote endpoints not unlike peer-to-peer systems. This allows for more flexible management of available resources during query execution, while still preserving the flexibility of a data federation.

Chapter 3

Limitation and Future Work

In the following we are going to detail the limitations of both AVALANCHE and its extension X-AVALANCHE, as well as those of the optimisation methods employed by both systems. In addition, based on these limitations and the findings of this thesis we will briefly discuss possible future work directions. The work presented in this thesis exhibits two kinds of limitations. First, both AVALANCHE and X-AVALANCHE could be extended and/or optimised further. Secondly, the external validity of the evaluation is limited.

6 System limitations and improvements

6.1 AVALANCHE

One of the major limitations of AVALANCHE stems from its lack of support for disjunctions or UNION SPARQL graph patterns. Because of this, the number of conjunctive plan fragments for some classes of queries, i.e., *semantically selective* queries, can be prohibitively large. As a direct consequence, AVALANCHE does not offer result-set completeness guarantees. Support for UNION graph patterns is implemented in AVALANCHE's extension: X-AVALANCHE, along with optimisation methods specifically designed to target large data partitioned setups.

Another limitation of the original AVALANCHE planner and execution engine is that the system can be resource wasteful. In order to keep the distributed state management protocol simple, AVALANCHE does not keep track of and reuse previously executed costly operations, i.e., joins. Keeping track of partial results can lead to resource scarcity especially for low selectivity queries. A future extension of AVALANCHE can benefit from investigating the applicability of methods like *Sideways Information Passing* SIP presented in [Neumann and Weikum, 2009a].

In addition, a high-impacting future work avenue consists of the enhancement of join cardinalities, one of the paramount issues characteristic to any centralised or distributed DBMS. To improve the accuracy and performance of join estimation, future work can benefit from investigating methods such as *sampling* [Estan and Naughton, 2006], *Characteristic Sets* [Neumann and Moerkotte, 2011] or *Graphical Models* [Tzoumas et al., 2012] to name a few.

Furthermore, in AVALANCHE we have only focused on Basic Graph Pattern (BGP) matching and have ignored SPARQL features like OPTIONAL and FILTER graph patterns. Possible future work on AVALANCHE can include support to cover these features. Providing adequate support for FILTER graph patterns is likely to improve

Avalanche's performance primarily due to the patterns increased selectivity (depending on how soon it can be evaluated), resulting in fewer partial results - a primary cause of performance degradation in distributed query processing.

6.2 Query Plan Fragmentation

When employing *plan fragmentation* to derive an optimal tradeoff between total query execution time and time to first results, the administrator is faced with a choice: either select a non-parametric space reduction method, e.g., *bayesian-blocks*, and allow for the automatic selection of the number of fragments given a predefined prior tuned to data distribution, or select the *k-segmentation* parametric method and face the further challenge in choosing a good value for the number of segments. A current limitation of the x-Avalanche extension is that this choice is not automated. Potential future work directions could include automatic learning of the parameters, i.e., which method to use and what is a good prior or number of segments, by investigating the applicability of methods like *Bayesian Optimisation* [Snoek et al., 2012] or other self-tuning database methods [Chaudhuri and Narasayya, 2007].

Furthermore, one more far-reaching limitation stems from the *impedance mismatch* between real and predicted plan performance. Like with any traditional DBMS, the optimality of any query plan is conditioned on the assumption that the cost model is accurate. However, this is rarely the case in reality. In consequence, fragmentation derived performance gains are diminished and depend on the estimative power of the cost model. Improving the cost model's accuracy will allow x-Avalanche to make better optimisation decisions and improve performance.

6.3 x-Avalanche

To further improve x-Avalanche's performance, a number of research avenues and potential solutions stand out. While x-Avalanche extends and enhances the distributed state management protocol of Avalanche, it does not address all sources of limitations. One such limitation is derived from the level of impact that low performing SPARQL endpoints have on the system. While this is addressed to a certain degree by caching result-sets, x-Avalanche does not cache SPARQL queries with VALUES bindings. A future extension could entail investigating how to use bloom filters [Broder and Mitzenmacher, 2003] to reduce the number of bindings sent to remote endpoints and therefore remote workload. Furthermore, the x-Avalanche union operator is not optimised to take duplicates into account. On the Web of Data records are duplicated leading to a more optimisation possibilities by investigating the applicability of bloom filters to this case or employing methods similar to the ones described in [Saleem et al., 2013].

Further research avenues to improving x-Avalanche include enhancements of the source selection process in order to reduce the number of unnecessary requests [Hose and Schenkel, 2012]. Finally, it is worthwhile to investigate how x-Avalanche can support the *schema messiness* of the Web of Data, by employing ontology similarities [David and Euzenat, 2008, David et al., 2010] to also execute equivalent query rewritings of the

original SPARQL query. This would allow x-Avalanche to consider similar datasources to the ones targeted by the original query and therefore to produce additionally relevant results.

7 Experimental Evaluation

The experimental setup for both Avalanche and x-Avalanche rely on a limited number of physical resources. In both cases a physical server has to accommodate more than a dozen SPARQL and Avalanche/x-Avalanche endpoints. When one machine accommodates multiple endpoints, then these endpoints compete for shared resources such as RAM, disk I/O, network I/O, and CPU-time. Our experimental setup is as realistic as possible in a laboratory-setup and allows for the generalisation of results. However, the inherent competition for resources can have a negative impact on measured system performance, a fact that can be mitigated by the choice of physical machines.

Chapter 4

Conclusions

In this thesis we address the question of how to efficiently query the Web of Data while taking into account its scale, diversity and unreliable and uncontrollable nature. We begin by first introducing Avalanche, a federated SPARQL engine which:

1. makes no assumptions about RDF data distribution to SPARQL endpoints,
2. is adaptive to changing network conditions, i.e, can adapt to slow network connections or endpoint unavailability,
3. retrieves up-to-date results from SPARQL endpoints, and
4. is flexible by making limiting assumptions about the structure of participating triple stores.

Tailored to address the *semantic heterogeneity* derived from the Web of Data's rich and broad semantic diversity, coupled with its characteristic lack of guarantees, Avalanche delivers partial results as they become available, favouring those that are faster to compose. While Avalanche puts more emphasis on low-latency results than on result-set completeness, the system is *eventually complete* if not stopped. Another important differentiating factor between Avalanche and other federated SPARQL engines is the fact that query-relevant statistical information is retrieved on-the-fly while querying. Furthermore, a loose-coupled federation engine, Avalanche supports *location transparency*, since the caller does not need to know where data is located. Being data distribution agnostic, Avalanche also supports *replication and fragmentation transparency*. Finally, Avalanche's architecture follows the principle of *decentralisation*, allowing any endpoint to assume the role of a query broker, a trait similar to peer-to-peer systems. Primarily IO-bound, the query broker supports asynchrony by using asynchronous HTTP requests to avoid blocking.

Our results show that by employing a fragmented and concurrent execution strategy, Avalanche is able to exhibit robustness and adaptivity in a changing environment like that of the Web of Data. By fragmented execution, we refer to the fact that the original SPARQL query is rewritten as the union of all *fragments* which comprise it. A *query fragment* is defined as the conjunction of all query triple patterns, where every triple pattern can be resolved by only one endpoint.

As the Web of Data continues to grow, we postulate that so is the likelihood that large numbers of endpoints will index data sharing the same vocabularies, forming *semantically homogenous* partitions of the Semantic Web. Focusing on this scenario and in order to address some of Avalanche's limitations, we introduce x-Avalanche an extension of our original system. Here, we add support for disjunctions by using a distributed union operator capable of scaling to hundreds or thousands of endpoints. We empirically

demonstrate that this operator exhibits a 5 fold performance increase over a naïve serial algorithm. Furthermore, we enhance Avalanche's distributed state management with:

- remote caches aimed to reduce the high latency typical of SPARQL endpoints, i.e., empirical evaluation results show that total query time, on average, dropped by 10% when using the Virtuoso v7.1 RDF store,
- multicast parallel bind-joins relying on the SPARQL 1.1 VALUES clause, and
- proxy based execution of x-Avalanche operators, i.e., offloading orchestration effort to participating endpoints not unlike peer-to-peer systems.

Finally, in x-Avalanche, we formalise and introduce a novel and parallel-friendly optimisation paradigm designed not only to offer an optimal tradeoff between total query execution time and fast first results, but also to consider an extended planning space unexplored so far. Consequently, x-Avalanche takes the *fragmented* execution model first introduced in Avalanche to its logical conclusion, by relaxing the restriction that for a query fragment a triple pattern can be answered by only one endpoint. Combined, x-Avalanche's enhancements and optimisations can lead to dramatic performance improvements over top performing state of the art federated SPARQL engines like FedX. Our results show that x-Avalanche can be up to 70 times faster when retrieving first results and up to 57 times faster for total query execution time, while being more than 20 times faster on average when resolving low selectivity (expensive) queries.

To conclude, the work presented in Avalanche and x-Avalanche shows that federated SPARQL processing can still be considerably improved by focusing on architectural enhancements, query optimisation as well as low-level operator design. In consequence, we believe that the insight gained from this work provides an important building block for further developing the Web of Data.

Part II

Contributions of this thesis

Chapter 5

Adaptive Federated SPARQL Querying

This chapter is based on a submission that was accepted at the:

Journal of Web Semantics:
Science, Services and Agents on the World Wide Web
volume 26, pages 1-28, 2014

Querying a Messy Web of Data with Avalanche

Cosmin A. Başca and Abraham Bernstein

DDIS, Department of Informatics, University of Zurich, Zurich, Switzerland
basca@ifi.uzh.ch, bernstein@ifi.uzh.ch

Abstract. Recent efforts have enabled applications to query the entire Semantic Web. Such approaches are either based on a centralised store or link traversal and URI dereferencing as often used in the case of Linked Open Data. These approaches make additional assumptions about the structure and/or location of data on the Web and are likely to limit the diversity of resulting usages.

In this article we propose a technique called Avalanche, designed for querying the Semantic Web without making any prior assumptions about the data location or distribution, schema-alignment, pertinent statistics, data evolution, and accessibility of servers. Specifically, Avalanche finds up-to-date answers to queries over SPARQL endpoints. It first gets on-line statistical information about potential data sources and their data distribution. Then, it plans and executes the query in a concurrent and distributed manner trying to quickly provide first answers.

We empirically evaluate Avalanche using the realistic FedBench data-set over 26 servers and investigate its behaviour for varying degrees of instance-level distribution "messiness" using the LUBM synthetic data-set spread over 100 servers. Results show that Avalanche is robust and stable in spite of varying network latency finding first results for 80% of the queries in under 1 second. It also exhibits stability for some classes of queries when instance-level distribution messiness increases. We also illustrate, how Avalanche addresses the other sources of messiness (pertinent data statistics, data evolution and data presence) by design and show its robustness by removing endpoints during query execution.

1 Introduction

With the advent of the Semantic Web, a Web-of-Data is emerging interlinking ever more machine readable data fragments represented as RDF documents or queryable semantic endpoints. It is in this ecosystem that unexplored avenues for application development are emerging. While some application designs include a Semantic Web data crawler, others rely on services that facilitate access to the Web-of-Data either through the SPARQL protocol or various APIs like the ones exposed by *Sindice*[1] or *Swoogle*[2]. As the mass of data continues to grow—Linked Open Data [Bizer et al., 2009a] accounts for 27 billion triples as of January 2011[3]—the scalability factor combined with the Web's uncontrollable nature and its heterogeneity will give rise to a new set of challenges. A question marginally addressed today is how to support the same messiness in querying the Web-of-Data that gave rise to the virtually endless possibilities of using the traditional Web. In other words: *How can we support querying the messy web of data whilst adhering to a minimal, least-constraining set of principles that mimic the ones of the original web and will—hopefully—support the same type of creative flurry?*.

[1] http://swoogle.umbc.edu/
[2] http://sindice.com/
[3] http://www4.wiwiss.fu-berlin.de/lodcloud/state/#domains

Translating the guiding principles of the Web to the Web-of-Data proposes that we should use a single communications protocol (i.e. HTTP with encoded SPARQL queries) and use a common data representation format (some encoding of RDF), which allows embedding links. In addition, it implicitly proposes that:

(a) we cannot assume any (or control the) distribution of data to servers,
(b) there is no guarantee of a working network,
(c) there is no centralised resource discovery system (even though crawled indices akin to Google in the traditional web may be provided),
(d) the size of RDF data no longer allows us to consider single-machine systems feasible,
(e) data will change without any prior announcement,
(f) there is absolutely no guarantee of RDF-resources adhering to any kind of predefined schema, being correct, or referring/linking to other existing data items—in other words: the Web-of-Data will be a mess and "this is a feature not a bug."

As an example, consider the life sciences domain: here information about drugs, chemical compounds, proteins and other related aspects is published continuously. Some research institutions expose part or all of their data freely as RDF dumps relying on others to index it as in the cases of the CheBi[4] and KEGG[5] datasets, while others host their own endpoints like in the case of the Uniprot dataset.[6] Hence, anybody querying the data will have:

- no control over its distribution, i.e. different copyright and intellectual property policies may prevent access to downloading part or the entire dataset but permit access to it on a per-query basis with potential restrictions like time and/or quota limits,
- no guarantees about the availability and network connectivity of the information sources, i.e. some institutions move repositories or change access policies, resulting in server unavailability,
- no guarantees about content stability as data changes continuously due to scientific breakthroughs/discoveries, and a plethora of schemas are used, i.e. some subdisciplines may favour dissimilar but overlapping attributes describing their results, have differing habits about using same-named attributes, and use a diversity of taxonomies with varying semantics.

Often-times problem domains and researchers' questions span across several datasets or disciplines that may or may not overlap. Even in the light of this messiness, the data about drugs, chemical compounds, proteins, and their interrelations is queried constantly resulting in a strong need to provide integrated and up-to-date (or current) information.

Several approaches that tackle the problem of querying the entire Web-of-Data have emerged lately, and most adhere to the explicit principles. They do, however, not address the implicit principles. One solution, *uberblic.org*,[7] provides a centralised queryable endpoint for the Semantic Web that caches all data. This approach allows searching for and joining potentially distributed data sources. It does, however, incur the significant problem of ensuring an up-to-date cache and might face crucial scalability hurdles in the

[4] http://www.ebi.ac.uk/chebi/
[5] http://www.genome.jp/kegg/
[6] http://beta.sparql.uniprot.org/
[7] http://platform.uberblic.org/

future, as the Semantic Web continues to grow. Additionally, it violates a number of the implicit principles locking-in data. Furthermore, as Van Alstyne *et al.* [Alstyne et al., 1995] argue, incentive misalignments would lead to data quality problems and, hence, inefficiencies when considering the Web-of-Data as "one big database."

Other approaches base themselves on the guiding principles of Linked Open Data publishing and traverse the LOD cloud in search of the answer. Obviously, such a method produces up-to-date results and can detect data locations only from the URIs of bound entities in the query. Relying on URI structure, however, may cause significant scalability issues when retrieving distributed data sets, since (i) the servers dereferenced in the URI may become overloaded and (ii) it limits the possibilities of rearranging (or moving) the data around by binding the id (*i.e.*, URI) to its storage location. Just consider for example the *slashdot effect*[8] on the traditional web. Finally, traditional database federation techniques have been applied to query the Web-of-Data. One of the main drawbacks with traditional federated approaches stemming from their ex-ante (*i.e.*, before the query execution) reliance on *fine-grained* statistical and schema information meant to enable the mediator to build efficient query execution plans. Whilst these approaches do not assume central control over data, they do assume ex-ante knowledge about it facing robustness hurdles against network failure and changes in the underlying schema and statistics (invalidating implicit principles b and f).

In this paper, we propose AVALANCHE, a novel approach for querying the messy Web-of-Data which (1) *makes no assumptions about data distribution, schema, availability, or partitioning* and is skew resistant for some classes of queries, (2) provides *up-to-date results* from distributed indexed endpoints, (3) is *adaptive* during execution adjusting dynamically to external network changes, (4) *does not require detailed fine-grained ex-ante statistics* with the query engine, and (5) is *flexible* as it makes limited assumptions about the structure of participating triple stores. It does, however, assume that the query will be distributed over triple-stores and not "mere" web-pages publishing RDF. The system, as presented in the following sections, is based on a first prototype described in [Başca and Bernstein, 2010] and brings a number of new extensions and improvements to our previous model.

Consequently, AVALANCHE proposes a novel technique for executing queries over Web-of-Data SPARQL endpoints. The traditional *optimise then execute* paradigm—highly problematic in the Web of Data context in its original conceptualisation—is extended into an exhaustive, concurrent, and dynamically-adaptive meta-optimisation process where fine-grained statistics are requested in a first phase of the query execution. In a second phase continuous query planning is interleaved with the concurrent execution of these plans until sufficient results are found or some other stopping criteria is met. Hence, the main contributions of our approach are:

– a querying approach over the indexed Web-of-Data, without fine-grained prior knowledge about its distribution

[8] http://en.wikipedia.org/wiki/Slashdot_effect

– a novel combination of interleaving cost-based planning (with a simple cost-model) with concurrent query plan execution that delivers first results quickly in a setting where join cardinalities are unknown due to lacking ex-ante knowledge
– a reference implementation of the Avalanche system

However, despite Avalanche's flexible and robust query execution paradigm, the method also comes with a set of limitations discussed in detail in Section 3. The main limitations are as follows:

– Avalanche does not benefit from the potential speedup exhibited by intra-plan parallelism since its current computation model does not support UNION-views,
– Avalanche can be resource wasteful for some classes of query workloads,
– embracing the WWW's uncertainties (see principles a-f), Avalanche neither guarantees result-set completeness nor the same result-set for repeated same-query executions.

Hence, Avalanche supports messiness stemming from the lack of ex-ante knowledge at various levels: data-distribution, schema-alignment, prior registration with respect to statistics, constantly evolving data, and unreliable accessibility of servers (either through network or host failure, HTTP 404's, or changes in policy of the publishers).

In the remainder we first review the relevant related work of the current state-of-the-art. The computational model is described in Section 3 while Section 4 provides a detailed description of Avalanche. In Section 5 we evaluate Avalanche against a baseline system (5.1), assess the query planner's quality (5.1), observe the system's behaviour when network latency varies (5.1) or when endpoints fail (5.1) and finally evaluate Avalanche with different data distributions (5.2) estimating the performance of our system. In Section 6 we present several future directions and optimisations, and conclude in Section 7.

2 Related work

Several solutions for querying the Web-of-Data over distributed SPARQL endpoints have been proposed before. They can be grouped into two streams: **I.** distributed query processing, **II.** RDF indexing, and **III.** statistical information gathering over RDF sources.

Distributed query processing: A broad range of RDF storage and retrieval solutions exist. They can be grouped along the dimensions of *partition restrictiveness* (i.e., the degree to which the system controls the data distribution) and the intended *source addressing space* (i.e., the design goal in terms of physical distribution of hosts from single machine through clusters and the cloud to a global uncontrolled network of servers) as shown in Figure 1. Although not intended as a measure of scalability and performance the Figure positions the various approaches relative to the desired goal – a globally addressable and highly flexible system: both paramount features when handling messy semi-structured data at large-scale.

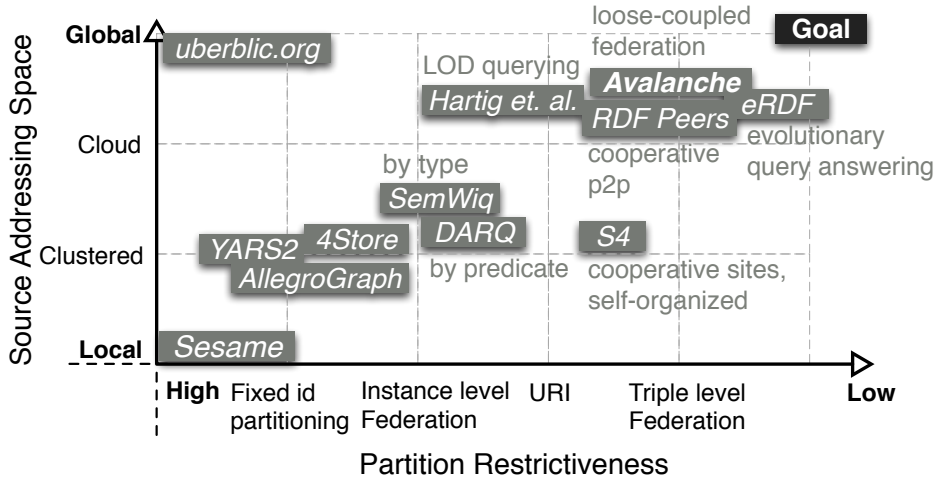


Fig. 1: Distributed SPARQL processing systems and algorithms, in relation to the desired goal (high flexibility & global addressing). This figure is not intended to provide an accurate positioning of the systems in the design space.

Research on distributed query processing has a long history in the database field [Kossmann, 2000, Sheth and Larson, 1990]. Its traditional concepts are adapted in current approaches to provide integrated access to RDF sources distributed on the Web-of-Data. For instance, *Yars2* [Harth et al., 2007] is an end-to-end semantic search engine that uses a graph model to interactively answer queries over semi-structured interlinked data, collected from disparate Web sources. Another example is the *DARQ* engine [Quilitz and Leser, 2008], which divides a SPARQL query into several subqueries, forwards them to multiple, distributed query services, finally, integrating the results of the subqueries. Inspired by peer-to-peer systems, *Rdfpeers* [Cai and Frank, 2004] is a distributed RDF repository that stores three copies of each triple in a peer-to-peer network, by applying global hash functions to its subject, predicate and object. Stuckenschmidt et. al [Stuckenschmidt et al., 2004] consider a scenario in which multiple distributed sources contain data in the form of publications. They describe how the *Sesame* RDF repository [Broekstra et al., 2002] needs to be extended, by using a special index structure that determines which are the relevant sources to be considered for a query. *Virtuoso* [Erling and Mikhailov, 2009]—a data integration software developed by OpenLink Software—is also focused on distributed query processing. The drawback of these solutions is, however, that they assume total control over the data distributions – an unrealistic assumption in the open Web.

Similarly, *SemWIQ* [Langegger et al., 2008] uses a mediator distributing the execution of SPARQL queries transparently. Its main focus is to provide an integration and sharing system for scientific data. Whilst it does not assume fine-grained control over the instance distribution they assume perfect knowledge about their `rdf:type` distribution. Addressing this drawback some [Schenk and Staab, 2008, Zemanek et al., 2008] propose to extend SPARQL with explicit instructions controlling where to exe-

cute certain sub-queries. Unfortunately, this assumes an ex-ante knowledge of the data distribution on part of the query writer. Finally, Hartig et al. [Hartig et al., 2009b] describe an approach for executing SPARQL queries over Linked Open Data [Bizer et al., 2009a] based on graph search. Whilst they make no assumptions about the openness of the data space, the Linked Open Data rules requires them to place the data on the URI-referenced servers – a limiting assumption for example when caching/copying data. A notable approach to browse the WoD and run structured queries on it is depicted by Sig.ma [Tummarello et al., 2010], a system designed to automatically integrate heterogenous web data sources. Suited to handle schema messiness Sig.ma differs from AVALANCHE mainly in its scope, which is that of aggregating various data sources in the attempt to offer a solution, while AVALANCHE (tackling data distribution messiness) does not integrate RDF indexes, but "guides" the query execution process to find exact matches.

Other flexible techniques have been proposed, such as the evolutionary query answering system *eRDF* by Guéret et. al [Guéret et al., 2008, Oren et al., 2008], where genetic algorithms are used to "learn" how to best execute the SPARQL query. The system learns each time a triple pattern gets executed. As the authors demonstrate, *eRDF* behaves better the more complex the query, while simple queries (one or two triple pattern queries) render low performance. Finally Muehleisen et. al [Muhleisen et al., 2010] advance the idea of a self organised RDF storage and processing system called *S4*. The approach relies on the principles of swarm-logic and exposes certain similarities with peer-to-peer systems.

RDF indexing: A number of methods and techniques to store and index RDF have been proposed to date, some like Hexastore [Weiss et al., 2008] and RDF3X [Neumann and Weikum, 2009a] construct on-disk indexes based on B+Trees while exploiting all possible permutations of *Subjects*, *Predicates* and *Objects* in an RDF triple. Other notable approaches include [Atre et al., 2010], where RDF is index using a matrix for each triple term pair – an approach suitable for low selectivity queries, suffering in performance however when highly selective queries are asked. Furthermore GRIN [Udrea et al., 2007] proposes a special graph index which stores "centre" vertexes and their neighbourhoods leading to lower memory consumptions and faster times to answer graph based queries than traditional approaches such as Jena[9] and Sesame[10].

Query optimisation: Research on query optimisation for SPARQL includes query rewriting [Hartig and Heese, 2007], join re-ordering based on selectivity estimations [Bernstein et al., 2007, Maduko et al., 2007, Neumann and Weikum, 2009a], and other statistical information gathering over RDF sources [Harth et al., 2010, Langegger and Woss, 2009]. *RDFStats* [Langegger and Woss, 2009] is an extensible RDF statistics generator that records how often RDF properties are used and feeds automatically generated histograms to *SemWIQ*. Histograms on the combined values of SPO (*S*ubject *P*redicate

[9] http://jena.sourceforge.net/
[10] http://www.openrdf.org/

*O*bject) triples have proved to be especially useful to provide selectivity estimations for filters [Bernstein et al., 2007]. For joins, however, histograms can grow very large and are rarely used in practice. Another approach is to precompute frequent paths (i.e., frequently occurring sequences of S, P or O) in the RDF data graph and keep statistics about the most beneficial ones [Maduko et al., 2007]. It is unclear how this would work in a highly distributed scenario. Finally, Neumann et. al [Neumann and Weikum, 2009a] note that for very large datasets (towards billions of triples) as even simple index scans become too expensive, single triple pattern selectivity is not enough to ensure accurate join selectivity estimation. As pattern combinations are more selective, they successfully integrate *holistic sideways information passing* with the recording of detailed join cardinalities of constants joined with the entire graph as means of improving join selectivity. An alternative approach is represented by summarising indexes as described by Harth et. al. [Harth et al., 2010] in *data summaries*.

3 Computational Model

Avalanche's computational model diverges from the traditional federated query processing paradigm in several key ways due to the uncertainties of the Web-of-Data (WoD) outlined above. In the following we will detail these characteristics, the assumptions from which they stem and the advantages and disadvantages they introduce while identifying some of the pertinent scenarios that Avalanche is suited for.

Guaranteeing *global completeness*—i.e., a complete result set (or answer set)—on the WoD is impossible due to its uncertainties. Servers may go down (or unreachable) at any given point in time not delivering triples necessary or new servers may appear on the but be unknown to the query engine. However, considering the restricted scope of the endpoints (or sources) selected to participate in a given query we advance the notion of result-set *query-contextual completeness*. By this we refer to the set of all tuples, which constitute the complete query answer if none of the participating endpoints fail.

For these reasons, in Avalanche we focus on optimising for answering SPARQL queries under uncertain conditions and constrains like the FAST FIRST limit modifier used in ORACLE RDB [Antoshenkov and Ziauddin, 1996]. Consequently, Avalanche is designed to deliver partial results as they become available favouring those that are faster to compose. If the query execution process is not stopped, Avalanche is *eventually complete* in the query-contextual scope. Hence, Avalanche puts more emphasis on the low latency part of the result-set than on completeness by allowing the query requester to specify various uncertain termination conditions (i.e., relative rolling saturation or first answers). In this sense, Avalanche behaves akin to a Web search engine where the first or most relevant results are fetched with the lowest attainable latency while initially ignoring the rest. Thus, Avalanche is suited for exploratory scenarios where the domain is unknown or changes often, situations where bulk data access is limited in some manner (i.e., legal or jurisdictional considerations), or scenarios where at least some results are required fast (i.e., to quickly render the first page with search results from a query).

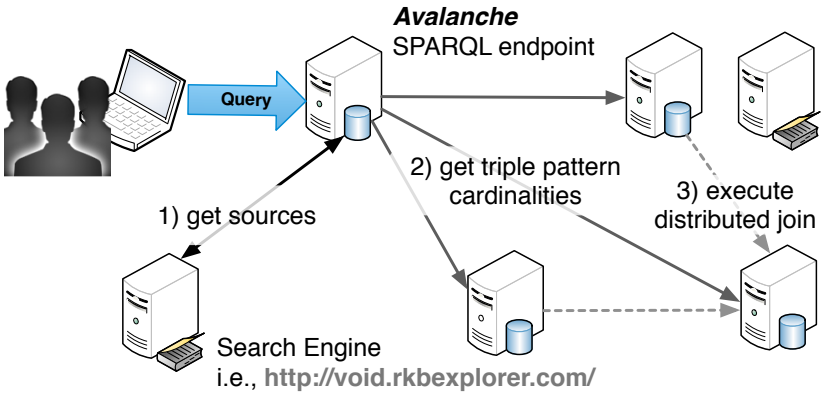


Fig. 2: A simplified view of the AVALANCHE execution model illustrating the three major phases: source discovery, statistics gathering, and query planing/distributed execution

A distributed query processing system, AVALANCHE splits the query execution process into three phases as seen in the diagram from Figure 2. The process closely resembles the traditional federated SPARQL processing pipeline: it first identifies the relevant sources to consider, it then retrieves fine-grained statistical information pertinent to the query being executed and finally resolves an optimised version of the original query.

Since finding the optimal plan for a distributed query is NP-hard solutions often rely on heuristics to find plans yielding higher levels of performance [Özsu and Valduriez, 1999]. In addition, further complications emerge due to the WoD's underlying uncertainties enumerated before. Hence, AVALANCHE introduces a number of changes to the querying process which depart from the traditional distributed query processing paradigm. In the remainder we discuss its characteristic heuristic and executions strategy.

Heuristics A heuristic that AVALANCHE employs when exploring the plan composition space is to consider only plans where *any triple pattern of the query can only be answered by one host*. This presents the following main advantages:

1) *generated plans are simpler* and therefore easier to optimise, i.e. using strategies like join-reordering,
2) *generated plans are easier to execute*, i.e., using traditional blocking join / merge physical operators – supported by a wider range of Semantic DBMS's, and
3) *the plan search space is reduced* since all possible plans where a triple pattern is bound to multiple hosts (combinatorial complexity) are not considered when estimating cost.

However, employing this planning heuristic, also introduces the following limitations:

i) *a high number of plans producing empty answer-sets* is generated for queries where the number of participating sites is much larger than the sites where partial results are located (i.e., highly localised queries that make use of widely used terminology),
ii) does not generate plans that contain *unions*.

Avoiding *unions* of partial results can be a severe limitation for some classes of queries while benefitting others. Consider for example the situation were a triple pattern can be answered by more than one host. The selectivity distribution of this triple pattern over selected sites can fall in one of the following situations: the triple pattern can be either *homogeneously selective* i.e., of comparable selectivity on all participating hosts or *heterogeneously selective* i.e., of varying (low and high) selectivity on participating hosts.
The *homogeneously selective* case is simpler since we can consider the union of all pertinent hosts for the given triple pattern. First, by doing so the number of generated plans is reduced by replacing all plans where the triple pattern was bound to one host with one plan that binds the triple pattern to all hosts. Second, the newly generated plan executes faster because it leverages the parallelism of the union operation. Finally the answer-set is larger because all hosts are considered as opposed to only one.
This is not the case when the triple pattern is *heterogeneously selective*. In this situation a union over all sites will severely hinder the performance of executing the plan due to the high latency and high resource utilisation of the high selectivity components of the union. Higher performance can be obtained for a subset of the results by considering only some of the hosts as participating in the union, at the expense of a combinatorial increase in the number of plans to search through.

Execution strategy AVALANCHE makes use of a concurrent execution strategy of all plans. Doing so confers the following advantages:

1) it has the potential to speed up query execution by leveraging inter-plan parallelism and by warming up local endpoint cache hierarchies, i.e. the same subquery is likely to be requested several times by different concurrent plans - with adequate concurrency control only the first request is executed while all subsequent ones are served from materialised memory views. This of course depends on available memory. For the same reason the execution of multiple *overlapping* queries could be sped up,

2) it attempts to mitigate the negative effect of empty answer-sets since the execution of plans that produce empty result-sets (*unproductive* plans) is intertwined with that of plans that produce non-empty answers (*productive* plans). Furthermore, *unproductive* plans are in general executed quicker since they can be halted early, when the first empty join is encountered.

Still, this execution strategy can be resource wasteful especially when multiple *non-overlapping* queries are executed. To address this, AVALANCHE makes use of various plan cost model heuristics when estimating plan cost in order to reduce resources wastefulness, essentially aiming to execute those plans deemed *productive* as early as possible. The plan generation process and cost estimation model are detailed in Section 4.

4 The Design and Implementation of an Indexed Web-of-Data Query Processing System

AVALANCHE is part of the larger family of Federated Database Management Systems or FDBMS's [Heimbigner and McLeod, 1985]. Focusing primarily on answering SPARQL queries over WoD endpoints, AVALANCHE relies on a commonly used data representation format: RDF and SPARQL as the main access operation. In contrast to relational FDBMS, where schema changes are costly and, therefore, happen seldom, the WoD is subjected to constant change, both schema and content-wise. In consequence, the major design contribution of AVALANCHE is that *it assumes the distribution of triples to machines participating in the query evaluation to be unknown prior to query execution.*

To achieve *loose coupling* AVALANCHE adheres to strict principles of transparency as well as heterogeneity, extensibility and openness. When submitting queries to an AVALANCHE endpoint the user does not need to know where data is actually located, ensuring *location transparency.* AVALANCHE endpoints are SPARQL endpoints that can additionally orchestrate the execution of queries according to the model we detail in the following sections. To achieve *replication and fragmentation transparency*, AVALANCHE is also data-distribution agnostic. In addition, participating endpoints are not constrained in any way with regard to the schemas, vocabularies, or ontologies used. Furthermore, over time the federation can evolve unrestrained as new data sources can be added without impacting existing ones.

Akin to peer to peer systems (p2p), AVALANCHE does not assume any centralised control. Any computer on the internet can assume the role of an AVALANCHE-broker. However, AVALANCHE is not a p2p system, since participating sites do not make fractions of their resources—*CPU, RAM*, or *disk*—directly available to other members, nor are they bookkeeping information concerning neighbouring hosts.

Another important distinction to existing federated SPARQL processing systems, lies within the early stages of the query execution. Traditionally, statistical information is indexed *ex-ante*, i.e., ahead of query execution time in the federation's meta-database from where it is later retrieved to aid the source selection and query optimisation processes. AVALANCHE relies on each participating site to manage their respective statistics individually – a trait shared to a varying degree by virtually any optimised RDF-store. Consequently, *query-relevant statistical information is retrieved at the beginning of each query execution phase* as illustrated in Figure 2.

In the following, we will first outline our approach, detailing its basic operators and the actual system using a motivating example. This will lead the way towards thoroughly describing the AVALANCHE components and its novelty.

4.1 System Architecture

The AVALANCHE system consists of the following major components working together in a concurrent asynchronous pipeline: (1) the AVALANCHE *Source Selector* relying on the *endpoints Web Directory* or *Search Engine*, (2) the *Statistics Requester*, (3) the

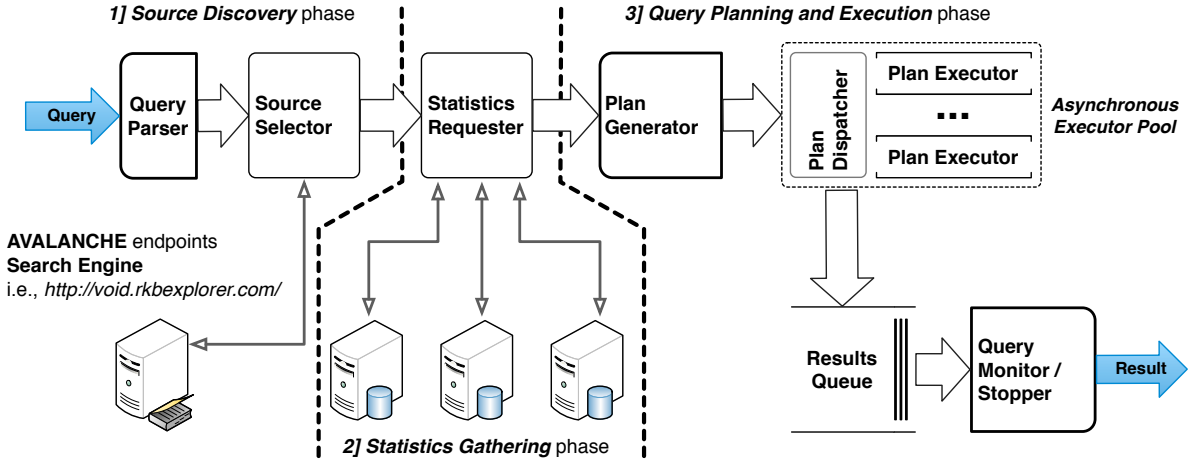


Fig. 3: The AVALANCHE execution pipeline

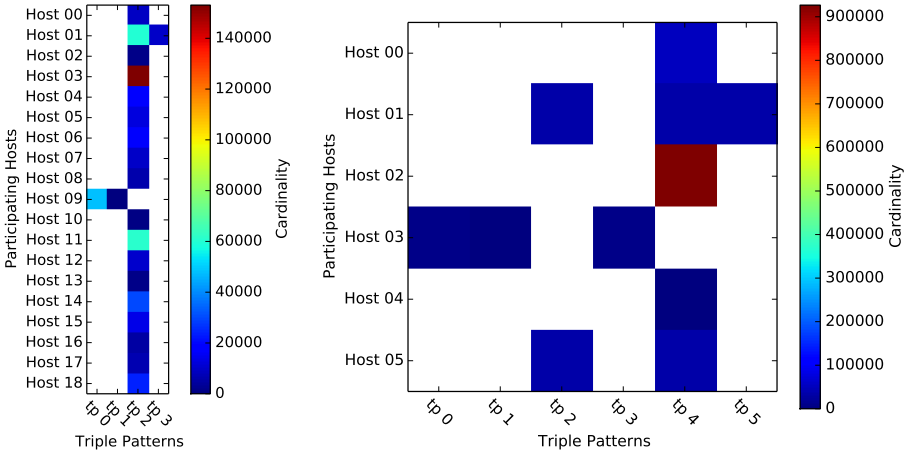
Plan Generator, (4) the *Plan Executor Pool*, (5) the *Results Queue* and (6) the *Query Execution Monitor/Stopper* as illustrated in Figure 3.

These components are coordinated into three query execution phases. First, participating endpoints are identified during the *Source Discovery* phase. Second, query specific statistics are retrieved during the *Statistics gathering* phase while finally followed by the *Query Planning and Execution* phase. We will now discuss how all the components are coordinated into these execution phases. The detailed technical description of the elements will be covered in the following subsections.

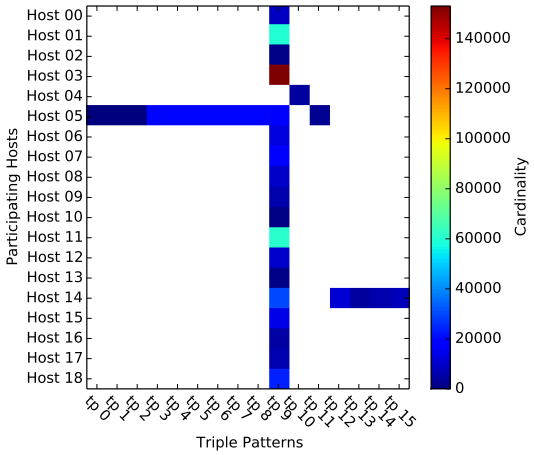
During *Source Discovery*, participating hosts are identified by the *Source Selector*, which interfaces with a *Search Engine* such as ***voID*** *store*,[11] *Sindice's*[12] SPARQL endpoint, or a *Web Directory*. A lightweight endpoint-schema inverted index can also be used. Ontological prefix (the shorthand notation of the schema, i.e. foaf) and schema invariants (i.e. predicates, concepts, labels, etc) are appropriate candidate entries to index. More complex source selection algorithms and indexes have been proposed [Li and Heflin, 2010] that could successfully be used by AVALANCHE given adequate protocol adaptations.

The next step—*Statistics gathering*—queries all selected AVALANCHE endpoints (from the set of known hosts H) for the individual cardinalities $card_{i,j}$ (number of instances) for each triple pattern tp_i from the set of all triple patterns in the query T_Q as detailed in Definition 1. The voID[13] vocabulary can be used to describe triple pattern cardinalities when predicates are bound or when schema concepts are used, along with more general purpose dataset statistical information, making use of terms like: void:triples, void:properties, void:Linkset, etc. Additionally, the same can be accomplished by using aggregating SPARQL COUNT-queries for each triple pattern or by simple specialised index lookups in some triple-optimised index structures [Weiss et al., 2008].

[11] http://void.rkbexplorer.com/
[12] http://sindice.com/
[13] http://www.w3.org/2001/sw/interest/void/



(a) $FQ5$ (b) $FQ14$



(c) $FQ33$

Fig. 4: Plan matrixes represented as heat-maps for selected Fedbench benchmark queries – for further details about the specific queries and benchmark please refer to Section 5.

Definition 1. *Given a query Q, T_Q is the set of all triple patterns $\in Q$ and H the set of all reachable hosts. $\forall tp_i \in T_Q$ and $\forall h_j \in H$, we define $\boldsymbol{card_{i,j} = card(tp_i, h_j)}$ as the **triple pattern cardinality** of triple pattern tp_i on host h_j.*

During the *Query Planning and Execution* phase, the *Plan Generator* proceeds with constructing the plan matrix (see Definition 2): a two dimensional matrix listing the cardinalities of all triple patterns gathered by the *Statistics Requester* (see Figure 3) of a query by possible hosts. Consider, for example, the plan matrixes for a selection of FedBench queries visualised in Figure 4 as a heat map, where white indicates the absence of triples matching a triple pattern tp_i on some host h_j (i.e., $card_{i,j} = 0$). Focusing on Figure 4 a) we, for example, see that only host-09 has triples matching tp_1.

Definition 2. *The matrix $\mathcal{PM}$ of size $|H|\times|T_Q|$ defined below is called the **plan matrix**, where the elements $card_{i,j}$ are triple pattern cardinalities as ascertained in Definition 1.*

$$\mathcal{PM} = \begin{bmatrix} card_{0,0} \cdots card_{0,n} \\ \vdots \quad \ddots \quad \vdots \\ card_{s,0} \cdots card_{s,n} \end{bmatrix} \tag{1}$$

The plan matrix is instrumental for the generation of query plans. Every query plan p contains one triple-pattern/host pair (tp_i, h_j) for each triple pattern tp_i in the query T_Q, where all tp_i match at least one triple (i.e., $card(tp_i, h_j) \neq 0$; see Definition 3). Thus, planning is equal to exploring the set of possible triple-pattern/host pairs resulting in valid plans. Visually, this corresponds to finding sets of non-zero cardinality squares, where each column is represented exactly once – the assumption that a triple pattern is bound to one host only.

Definition 3. *A **query plan** is the set $p = \bigcup(tp_i, h_j)$ that contains exactly one triple-pattern/host-pair (tp_i, h_j) per $tp_i \in T_Q$, where $card(tp_i, h_j) \neq 0$ and $h_j \in H$.*

While some queries can produce no plans, the universe of all plans (see Definition 4) has a theoretical upper-bound equal to $|H|^{|T_Q|}$, however the exact number of plans constructed according to our computational model can be derived using equation 2. Albeit an exponential number of possible plans can theoretically exist, our empirical evaluation suggests that real-world datasets often produce sparse plan matrixes—possibly a consequence of the LoD's heterogeneity—resulting in a significantly lower number of valid plans (i.e., akin to the plan matrixes in Figure 4). Hence, the task of the *Plan Generator* is to explore the space of all possible valid SPARQL 1.1 rewritings of the original query Q by pairing triple patterns from T_Q with available endpoints from H, under the assumption that a triple pattern is bound only to one host. Therefore, unlike traditional DBMS's Avalanche generates *incomplete plans* i.e., where each plan in isolation cannot guarantee result set completeness.

Definition 4. *The set of all plans for query Q, $P_Q = \{p_i \mid p_i$ is a query plan as in Definition 3 $\}$ is called the **query plan space** or **universe of all plans**.*

$$|P_Q| = \prod_{tp_j \in T_Q} |\{h_i \mid \text{iff } card_{i,j} \neq 0\}|, \quad 0 \leq |P_Q| \leq |H|^{|T_Q|} \tag{2}$$

It is important to note that factors such as the sheer size of the Web-of-Data, its unknown distribution, and multi-tenancy aspect may prevent Avalanche from guaranteeing result completeness. Whilst the proposed planning system and algorithm are complete, the execution of all plans to ensure completeness could be prohibitively expensive. Hence, Avalanche will normally not be allowed to exhaust the entire search

space—unless the query is simple or the search space is narrow enough. Consequently, AVALANCHE will try to optimise the query execution to quickly find the **first K** results by first picking plans that are more "promising" in terms of getting results quickly.

As soon as a plan is found, it gets dispatched to be handled by one of the *Plan Executor and Materialiser* workers in the *Executors Pool*. All workers execute concurrently. When a plan finishes, the executor worker places its results, if any, in the *Results Queue*–the queue is continuously monitored by the parallel running *Query Monitor* to determine wether to stop the query execution. Worker slots in the *Executors Pool* are assigned to new workers / plan pairs as soon as plans are generated and slots are available. If the pool is saturated, plans are queued until a worker slot becomes available again. To further reduce the size of the search space, a windowed version of the search algorithm can be employed. Here only the first P partial plans are considered with each exploratory step, thus sacrificing completeness.

In order to optimise execution, AVALANCHE employs both a common ID space and a set of endpoint capabilities, which we succinctly discuss in the following.

Common IDs A requirement for executing joins between any two hosts is that they share a common *id* space. The natural identity on the web is given by the URI itself. However some statistical analyses of URIs on the web[14] show that the average length of a URI is 76 characters, while analyses of the Billion Triple Challenge 2010 dataset[15] demonstrate that the maximum length of RDF literals is 65244 unicode characters long with most of the string literals being 10 characters in length. Therefore, using the actual RDF literal constants (URIs or literals) can lead to a high cost when performing distributed joins. To reduce the overhead of using long strings we used a number encoding of the URIs. To avoid central points of failure based on dictionary encoding or similar techniques, we propose the use of a hash function responsible for mapping any RDF string to a common number-based id format. For our experiments, we applied the widely used SHA family of hash functions on the indexed URIs and literals. An added benefit of a common hash function is that the hosts involved in answering a query, can agree on a common mapping function prior to executing the query. Note that this proposition is not a necessary condition for the functioning of AVALANCHE but represents an optimisation that will lead to performance improvements.

Endpoint operations To optimise SPARQL execution performance AVALANCHE takes advantage of a number of operations that extend the traditional SPARQL endpoint functionality. Whilst we acknowledge that the implementation of these procedures puts a burden on these endpoints their implementation should be trivial for most triple-stores. Some of the operations are either SPARQL 1.1 compliant or can be expressed as plain SPARQL queries, like getting triple pattern cardinalities, total number of triples or executing subqueries which are fully detailed in A, while others will be internally available in any indexed triple store and "only" need to be exposed (i.e. *set filtering*

[14] http://www.supermind.org/blog/740/average-length-of-a-url-part-2
[15] http://gromgull.net/blog/category/semantic-web/billion-triple-challenge/

or *set merge*). From a functional point of view the procedures are classified into two *execution operators* and *state management operators*.

The next subsections will describe the basic AVALANCHE operators and the functionality of its most important elements: the *Plan Generator* and *Plan Executor / Materialiser* as well as will explain how the overall execution pipeline stops.

4.2 Query Optimisation

To contextualise AVALANCHE further, consider the example query $Q_{example}$ in Listing 6.1, executing over the Fedbench[16] benchmark datasets. Specifically the query requests data that are distributed across three life-sciences domain datasets: DrugBank,[17] KEGG,[18] and ChEBI[19]. It is AVALANCHE's goal to find all drugs from DrugBank, together with their URL from KEGG and links to their respective graphical depiction from ChEBI.

Traditionally, query optimisers perform an exhaustive search of the plan universe in order to find the "best" plan given a set of optimisation criteria. The long established *dynamic programming* method is used for this purpose. To further reduce the cost of finding the best plan, the search space is pruned heuristically. A popular heuristic when doing so is to discard all plans with the exception of left-deep ones. Even in the light of these optimisations, exhaustive strategies for traversing the entire plan universe in order to find the best (or lowest cost) plan can become prohibitively expensive for queries where the number of joins is high, i.e. as reported in [Ramakrishnan and Gehrke, 2003] a number of 15 joins was considered prohibitive circa 2003. Moreover, when dealing with *uncertain constraints* such as FAST FIRST results, RDBMS's like Oracle RDB [Antoshenkov and Ziauddin, 1996] heuristically execute several plans competitively in parallel for a short interval of time to increase the likelihood of hitting the most relevant cases under the assumption of a ZIPF distribution.

```
1  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2  PREFIX drugbank: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/>
3  PREFIX chebi: <http://bio2rdf.org/ns/bio2rdf#>
4  PREFIX dc: <http://purl.org/dc/elements/1.1/>
5  SELECT ?drug ?keggUrl ?chebiImage
6  WHERE {
7   ?drug rdf:type drugbank:drugs .
8   ?drug drugbank:keggCompoundId ?keggDrug .
9   ?drug drugbank:genericName ?drugBankName .
10  ?keggDrug chebi:url ?keggUrl .
11  ?chebiDrug dc:title ?drugBankName .
12  ?chebiDrug chebi:image ?chebiImage .
13 }
```

Listing 6.1: Contextualising example - Life Sciences query from the Fedbench benchmark.

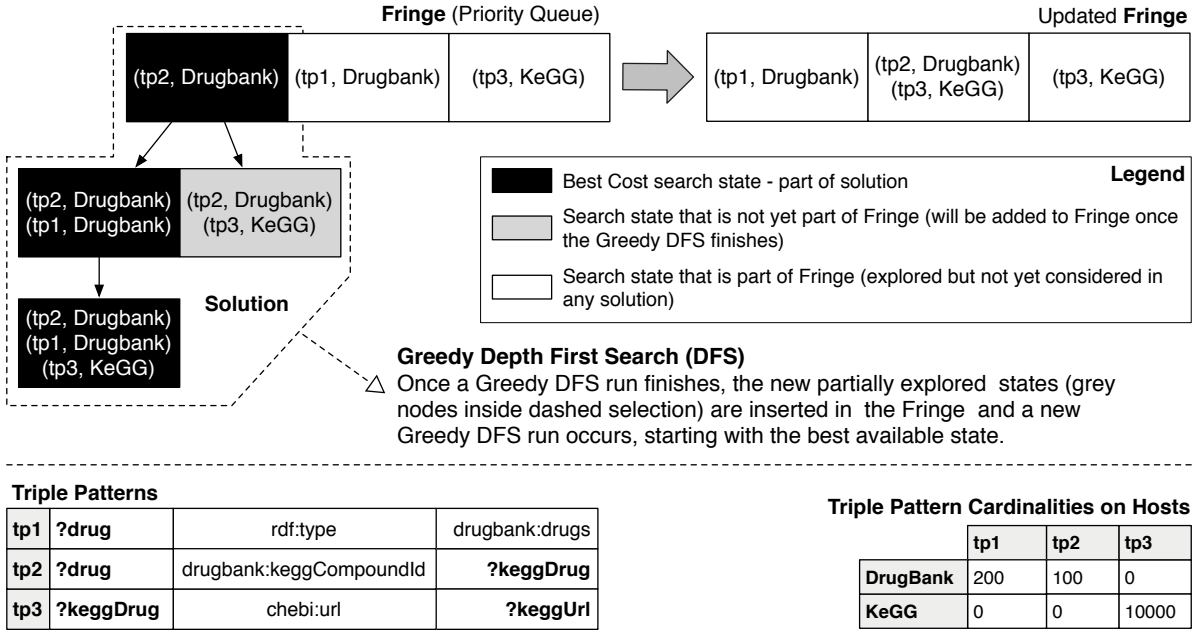
[16] https://code.google.com/p/fbench/
[17] http://www.drugbank.ca/
[18] http://www.genome.jp/kegg/
[19] http://www.ebi.ac.uk/chebi/

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX drugbank: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/>
PREFIX chebi: <http://bio2rdf.org/ns/bio2rdf#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
SELECT ?drug ?keggUrl ?chebiImage WHERE {
 SERVICE <http://drugbank-endpnt/sparql> {
  ?drug drugbank:genericName ?drugBankName .
  ?drug drugbank:keggCompoundId ?keggDrug .
  ?drug rdf:type drugbank:drugs
 } .
 SERVICE <http://chebi-endpnt/sparql> {
  ?chebiDrug chebi:image ?chebiImage .
  ?chebiDrug dc:title ?drugBankName
 } .
 SERVICE <http://kegg-endpnt/sparql> {
  ?keggDrug chebi:url ?keggUrl
 }
}
```

Listing 6.2: Motivating example query rewritten as a SPARQL 1.1 federated query.

Given that WoD SPARQL endpoints are not under any form of centralised control and network / system failures can occur any time, guarantees about the completeness of a SPARQL query answer cannot be claimed. Consequently, in AVALANCHE we focus on optimising for uncertain constrains akin to the FAST FIRST limit used in Oracle RDB. To this end, AVALANCHE performs an exhaustive search of the plan universe similar to traditional optimisers, with one critical difference: as soon as a plan is generated it is dispatched for execution while the optimiser continues to generate plans. As a first cost-reducing heuristic, we consider only plans where each triple pattern is assigned to one endpoint only. Therefore, each plan is equivalent to a SPARQL 1.1 decomposition of the original query without considering UNION graph patterns. For example one such plan (or decomposition) can be seen in Listing 6.2, where the SERVICE clause is used to bind triple patterns to endpoints.

Plans (or decompositions) can be classified into two categories: *productive plans* – those for which results are found – and *unproductive plans* – those for which no results are found. Considering this, just like in Oracle RDB we adopt the assumption that *the concurrent execution of plans will have a higher probability of yielding results if productive plans are found and dispatched early by the planer*. Hence, AVALANCHE also executes plans in parallel with the notable difference to Oracle RDB that it sets out to execute all plans until results are found or the stopping criteria are met. As a result the order in which plans are generated is critical, since this is the order in which they are also executed. As our empirical results from Section 5.1 show first results are found early during plan generation and execution. For many of the benchmark queries first results also coincide with total query results. A disadvantage of this approach is the apparent wasting of resources. We alleviate this problem by extending the SPARQL endpoint functionality with stateful distributed join processing by caching partial results in memory for the duration of the entire query. In this manner, when the same subquery is



Triple Patterns

tp1	**?drug**	rdf:type	drugbank:drugs
tp2	**?drug**	drugbank:keggCompoundId	**?keggDrug**
tp3	**?keggDrug**	chebi:url	**?keggUrl**

Triple Pattern Cardinalities on Hosts

	tp1	**tp2**	**tp3**
DrugBank	200	100	0
KeGG	0	0	10000

Fig. 5: Graphical example of a snapshot of the plan-generator traversal algorithm for a simplified version of $Q_{example}$. For brevity only three triple patterns are considered from $Q_{example}$ while the plan-generator algorithm is detailed over the first step.

part of multiple plans on the same endpoint, the effort of retrieving results from disk is spent only the first time. Furthermore, we assume that expensive and unproductive plans, which would consume resources needlessly, are discarded early by local endpoint optimisers – a feature supported by most industrial-strength RDF stores.

One of the main advantages conferred by this approach is that it relaxes the need for near-exact plan cost estimation. While for traditional query optimisers it is critical to estimate the cost as best as possible because only one plan (the best) is executed, in Avalanche since all plans are executed concurrently the best plans need only be ranked towards the beginning of the execution chain. Hence, the focus falls on the relative ranking of plans to each other. To generate plans efficiently the plan generator has to meet the following criteria:

- it must *generate plans in an order that matches as much as possible the order given by their estimated cost*, with the lowest cost estimate first, and
- *construct plans in an iterative fashion*, since waiting for an exhaustive composition of all plans is expensive – see Definition 4 for the upper bound.

Considering these requirements, we created a new graph traversal algorithm which we call: *Priority Queued Greedy DFSs*. The algorithm toggles between two modes of operation. First, it starts by seeding the global fringe implemented by a priority queue with all combination of triple pattern - endpoint pairs. Second, a localised *Greedy DFS* is performed starting with the best (or lowest cost) state from the global fringe i.e., node $(tp2, Drugbank)$ in Figure 5. From this point on, expansion is performed using a local

fringe, implemented by a stack. Nodes are pushed to the stack in order of their depth and for each depth level in order of their cost estimate. After a solution node is found i.e., ((tp2, Drugbank), (tp1, Drugbank), (tp3, KeGG)), the local fringe is inserted into the global fringe. The local Greedy DFS ensures the second criteria, while the global fringe ensures that multiple DFS searches can be performed efficiently because of the inclusion of partially explored solutions i.e., the grey node $((tp2, Drugbank), (tp3, KeGG))$ in Figure 5. We detail the plan generator algorithm in Section 4.4.

4.3 The Cost Model

Commonly, cost models can be classified into cost models that either aim to reduce the total time to execute the query or strive to reduce the response time or first result(s) latency. The first class of cost models are in general pertinent to single query execution scenarios. Since a complete result set is not in AVALANCHE's scope the second class of cost functions is desirable. Unlike the comprehensive cost model highlighted by Ozsu and Valduriez in [Özsu and Valduriez, 1999] AVALANCHE features a more relaxed cost model since it does not aim at producing one single cost-optimal plan but instead aims to execute all plans concurrently. Note that in practice concurrency is limited to a number of concurrent operations, a parameter chosen by the administrator (DBA) in line with the desired / possible load of the underlying broker/endpoint hardware. In consequence, since AVALANCHE needs to rank all generated plans as close as possible to the order of their cost estimates, two simplifying assumptions can be considered:

- *Network*: We assume that network latency and bandwidth are relatively uniformly distributed between participating sites. Although a gross approximation, the assumption holds true in most cases for geographically "near" sites. Furthermore, many participants on the WWW follow this assumption.
- *Distributed Joins*: A widely encountered phenomenon on the WoD, multi-tenancy gives rise to a number of difficulties and problems ranging from management of RDF data to query and index optimisation both locally and at a global scale. Since AVALANCHE's scope is the indexed WoD, it is unrealistic to assume that full index statistical information is always available or can always be shared between participating sites. Therefore, in the absence of more exact and elaborate metrics join selectivity is estimated. The main advantages of this model are: *1*) there is no need for joint distribution statistics to be available and *2*) it bears virtually no computation and network cost. However, there are many fallacies introduced as it offers no guarantees regarding the size of the join between any two BGPs.

In the following we discuss the impact these assumptions have on the cost model.

Selectivity estimation In the absence of exact statistics (i.e., join cardinalities) regarding *triple patterns* and *basic graph patterns*, selectivity is usually estimated. However, as AVALANCHE starts with the premise that triple pattern cardinalities are know as reported by `getTPCardinality` (A), triple pattern selectivities are computed and not

estimated. For a given triple pattern ***tp*** bound to a given host ***h*** its selectivity represents the probability of selecting a triple that matches from the total number of triples involved and is thus directly computed as follows:

$$sel^h_{tp} = P_{match}(tp, h) = \frac{card(tp, h)}{T_{MAX}} \tag{3}$$

where $T_{MAX} = \sum_{i=0}^{|H|} triples_{h_i}$, with $triples_{h_i}$ representing the total number of triples on host h_i.

Most RDF database management systems (with very few exceptions [Neumann and Weikum, 2009a]) *estimate* the selectivity of BGPs. In doing so AVALANCHE discriminates between *star* shaped graph patterns and the rest. Graph theoretic constructs, *star* graph patterns, materialise in the realm of SPARQL queries as groups of triple patterns that join on the same subject or object. For simplicity we will later refer to them as *star graph patterns* or *stars*. Any given basic graph pattern *bgp* can be decomposed into the set of all contained stars referred to as $\mathbb{S}_{bgp}$ and a remainder graph pattern which contains all triple patterns that do not form stars called $\mathbb{NS}_{bgp}$. In consideration of the above, the selectivity of *bgp* is estimated according to the the following formula:

$$SEL^h_{bgp} = \prod_{tp' \in \mathbb{NS}_{bgp}} sel^h_{tp'} \times \prod_{star \in \mathbb{S}_{bgp}} (\min_{tp'' \in star} sel^h_{tp''}) \tag{4}$$

The equation captures the intuition that non-star pattern triple-patterns are estimated via independent combination of their selectivities. Obviously, independence is not correct but oftentimes found as an acceptable approximation. The selectivity of a star pattern, in contrast, is estimated by the selectivity of its minimal participating triple-pattern.

Cost model When ranking plans, AVALANCHE employs a common no-preference multi-objective optimisation method: the method of *Global Criterion* [Zeleny, 1973]. AVALANCHE uses this method as an envelope to combine the following heuristic objectives:

a) **plan selectivity estimation**: this objective relies primarily on selectivity estimation as it appears in equations 3 and 4 and is defined according to the following equation:

$$SEL_{plan} = \prod_{sq \in SQ_{plan}} SEL^{h_{sq}}_{bgp_{sq}} \tag{5}$$

where *plan* represents a partial or complete plan and SQ_{plan} is the set of subqueries in *plan*.

b) **number of subqueries**: stemming from a *data-locality* assumption (related assertions are usually on the same host) this second heuristic is intended to bias the plan

generator towards plans (or partial plans) that will result in query decompositions with fewer subqueries and is defined as follows:

$$SIZE_{plan} = |T_{plan}| - |SQ_{plan}| \tag{6}$$

where $plan$ represents a partial or complete plan, $T_{plan} = \{tp_i \mid tp_i \in plan\}$ is the set of triple patterns in $plan$, and SQ_{plan} is the set of subqueries in $plan$.

Since AVALANCHE needs to compare partial plans with various degrees of completion whilst exploring the universe of all plans P_Q the number of subqueries is "normalised" by the number of triple patterns considered so far. Additionally, since the method of global criterion is sensitive to the scaling of the considered objective functions, as recommended in [Miettinen, 1998], the objectives are normalised into the uniform [0,1] range. Finally, AVALANCHE minimises the cost of a plan by combining the previous heuristic functions according to the following equation:

$$COST_{plan} = ||< SEL_{plan}, SIZE_{plan} > - z^{ideal}|| \tag{7}$$

where z^{ideal} represents the ideal or target cost value and the $||.||$ norm is the L_2 norm or the euclidean norm.

One of the main advantages of the cost model defined in this manner, is the flexibility conveyed by the fact that new heuristics can easily be plugged in. Plugging-in an additional element to the cost function would entail extending the cost vector $< SEL_{plan}, SIZE_{plan} >$ with an additional performance indicator as well as z^{ideal} with the desired target value for this indicator. We chose to favour high selectivity plans first over low selectivity ones, mainly due to the assumption that in general they are less costly to execute, thus reducing the time / resource usage penalty in case no results are found. Low selectivity plans are not discarded altogether, but simply given lower priority during execution. Hence, the target value for the first element of the cost function is 0. In addition, the second objective favours plans with fewer distributed joins (fewer subqueries) subscribing to a similar rationale: they are often cheaper to execute by pushing complexity towards local endpoints while avoiding expensive network transfers and connections – a fact particularly detrimental for queries that produce few results. Consequently the target value of the second element of the cost function is also 0 resulting in $z^{ideal} =< 0, 0 >$. Hence, for these these two performance indicators z^{ideal} could be omitted from the formula. This would, however limit the generality of the cost function, as elements with target values other than zero could not be added.

4.4 Plan Generation

As seen in Algorithm 1, the planner will try to optimise the construction of all plans using an informed repeated greedy depth traversal strategy. Due to its repeated nature, plans are not generated in strict ascending order of their estimated cost. Instead they are

generated in a partially sorted order primarily dictated by the partial cost estimates from the exploration fringe $\mathbb{F}$. This is achieved by minimising the *cost-estimation* function of each plan $COST_{plan}$, described in Equation 7. As designed, the plan generator's worst case complexity is $O(m^n)$.

Algorithm 1 Plan Generation

Precondition: Q a well-formed SPARQL query, $\mathbb{T}$ the set of all triple patterns $\in Q$
Postcondition: $\mathcal{N}$ a set of search nodes, P a query plan

```
1:  procedure planGenerator(Q)
2:      𝕍 ← ∅                                          ▷ 𝕍: set of visited nodes
3:      𝒞 ← ∅                                          ▷ 𝒞: set of closed nodes
4:      𝔽 ← nodes(𝕍, 𝕋, ∅)                             ▷ 𝔽: active exploration fringe
5:      ρ ← 0                                          ▷ ρ: current plan counter
6:      while 𝔽 ≠ ∅ do
7:          if ρ = MAX_plans then
8:              break
9:          best ← 𝔽.pop()                             ▷ best is a leaf search node
10:         if isSolution(best) then
11:             emit Plan(Q, best, ρ)                  ▷ emit newly found solution as a plan
12:             ρ ← ρ + 1
13:             𝔽.sort()                               ▷ sort fringe 𝔽 based on COST
14:         if best ∉ 𝒞 then
15:             𝒞 ← 𝒞 ∪{best}
16:             T_nxt ← {tp}, tp ∈ 𝕋 ∧ tp ∉ triplePatterns(best)
17:             if T_nxt = ∅ then                      ▷ T_nxt: next unexplored triple pattern in partial plan
18:                 continue
19:             𝔽 ← 𝔽 ∪ nodes(𝕍, T_nxt, best)          ▷ expand search space
20:
21: function nodes(V, T, parent)                       ▷ local fringe expansion function
22:     𝒩 ← ∅                                          ▷ 𝒩: the nodes, V: visited queue, T: a set of triple patterns
23:     for tp ∈ T do
24:         for h ∈ H do                               ▷ H: the set of all endpoints
25:             n ← node(tp, h, parent)                ▷ create a new search node for tp and h
26:             if n ∉ V ∧ n ≠ ∅ then
27:                 V ← V ∪ {n}
28:                 𝒩 ← 𝒩 ∪{n}
29:     𝒩.sort()                                       ▷ sort local fringe 𝒩 based on COST
30:     return 𝒩
```

With each exploratory step the size of the global fringe $\mathbb{F}$ increases by the number of sites $|H|$ (line 19). This happens for each expanded state or partial plan represented by a $< tp_i, h_j >$ pair, where $tp_i \in T_Q$ is the current triple pattern and $h_j \in H$ a participating endpoint or host. Not considering pruning, the algorithm is *complete* and *exhaustive* as it iterates over all possible plans. While traditional optimisers stop and return when the optimal *solution* is found, the planGenerator procedure is not halted and instead each solution or plan is **emitted** to the caller (line 11). The generator procedure is in essence a repeated application of a Greedy Depth First Search algorithm driven by a priority-queue-based fringe, which keeps track of all partial plans explored so far. This ensures that search states (or partial plans) are not visited multiple times. The Greedy

DFS aspect is necessary to produce viable plans quickly and is encoded by the partial sort of the local fringe $\mathcal{N}$ in function NODES (line 29). Here the exploration of direct descendant partial plans of the current state is enforced. In contrast, the global fringe $\mathbb{F}$ re-sorts (for efficiency we use a heap) all the partial plans explored so far from all previous Greedy DFS runs (line 13). This is critical since the planner must select for expansion the next best plan available.

Pruning As the exploration space grows quickly, pruning invalid or $\emptyset$ plans is desired. Early pruning is achieved immediately after the *statistics gathering* phase when the plan matrix $\mathcal{PM}$ is available, by removing all hosts (matrix rows) for which the cardinality of all triple patterns is 0. In the absence of triple-patten cardinalities, early pruning would not be possible and the maximum number of plans would have to be considered: $|H|^{|T_Q|}$. Hence, queries that produce a $0_{|H|,|T_Q|}$ plan matrix (zero matrix) are stopped during this early optimisation step.

Furthermore, during execution the same join can be often shared by multiple competing plans. Consequently, joins that are $\emptyset$ (empty) are recorded and used as dynamic feedback for the planner, which then prunes any plan that contains an $\emptyset$ join. This aspect transforms the AVALANCHE planner into an *adaptive* planner as seen in line 26 of the NODES function.

4.5 Query Execution

As we stated in the previous sections, AVALANCHE conceptually sets out to execute all plans concurrently. In practice however this can lead to high system load when queries are large (number of triple patterns) and have partial results on many endpoints. In the following we will describe how this problem is addressed in our system. Since any AVALANCHE endpoint can play both the role of a query broker and a SPARQL endpoint, in order to differentiate between the two roles we will simply refer to the endpoint which orchestrates the distributed execution of the query as *Query Broker* while referring to the rest simply as endpoints. Plans are dispatched for execution given the partially sorted order of their cost estimates. Since AVALANCHE optimises for FAST FIRST results, fast executing plans are favoured. If no stopping criteria is specified (i.e., LIMIT, timeout, etc) and participating endpoints maintain their availability, AVALANCHE finds all results every time a query is executed under these conditions, albeit in different orders if no explicit sort is specified. However, since no guarantees can be claimed in a multi-tenant setup like the WoD, due to the unpredictability of external factors, AVALANCHE looses its deterministic query resolution.

Addressing the Query Broker system load Once the triple pattern cardinalities are retrieved and the plan matrix $\mathcal{PM}$ constructed, the Query Broker is primarily responsible with three tasks, as seen in Figure 3: plan generation, plan execution orchestration and query progress monitoring–to determine when to stop. Except for plan generation, all other tasks are mainly I/O bound. We optimise the plan generation algorithm by making use of *memoization* to store the cost of partially constructed plans while traversing

the plan space. The plan execution orchestration process is centered around the *Executors Pool*. Considering its I/O bound nature, an *evented* socket-asynchronous paradigm is a natural fit. Using an event loop driven pool instead of a thread pool when dealing with I/O bound tasks can lead to dramatic improvements in terms of the number of concurrent tasks that can be handled at a fraction of the resources used otherwise. While we cannot directly compare to a thread based pool (i.e., due to implementation impedance mismatches which would result in increased development costs), anecdotal evidence suggests that evented task processors can potentially process several orders of magnitude more tasks than thread based ones, if tasks are non-blocking (e.g., I/O requests). Therefore, we based the implementation of the *Executors Pool* on the popular `libevent`[20] event loop.

Addressing Endpoint system load While the Query Broker can drive many plans concurrently due to its asynchronous architecture, the system load of participating query endpoints can still be high. We employ two strategies to reduce this burden on query answering endpoints. First, not all plans are dispatched for concurrent execution at the same time but instead a concurrency limit is set on the *Executors Pool*–similar to the number of worker threads in standard thread-pools, but featuring more workers. Currently, this parameter has to be set manually by the system administrator in concordance to available Query Broker system resources or desired load. Second, each endpoint caches the partial results of each received subquery in memory. Since each plan is executed in order of the selectivity estimation of its composing subqueries, the size of partial results (number of tuples) is kept as low as possible. Clearly, this reduces the cost of executing remote subqueries particularly when the same subquery is requested by multiple plans. This is typically the case when some RDF statements are located on only one site and can be joined with more RDF fragments from other endpoints. In addition, each AVALANCHE endpoint is enhanced with distributed join processing capabilities, also implemented using the same asynchronous evented task processing paradigm.

Plan Execution As soon as a plan is assigned to a worker, the process described in Algorithm 2 unfolds. Figure 6 illustrates this process for the query $Q_{example}$.

A first step consists of sorting the subqueries (if more than 1) in order of their selectivity estimation SEL^h_{sq} on the designated host h. The distributed join is then executed in left-deep fashion, starting with the most selective subquery, as seen in line 6 and steps 1 and 2 in Figure 6. Necessary for the next phase, the order in which joins occurred is recorded in the J_Q queue. The next phase is optional, since it's an optimisation. When enabled, the partial results that have been produced in the earlier join can be reconciled (filter out the pairs that do not match on the remote site) in reverse order of their counter-part joins (line 7, steps 3,4 in figure). Reconciliation can be naive (send the entire set compressed or not) or optimised. The former is used when the cost of creating the optimised structures is higher than just sending the set. In the latter hashes can be send when the item size is larger than its hash or following [Ramesh

[20] http://libevent.org/

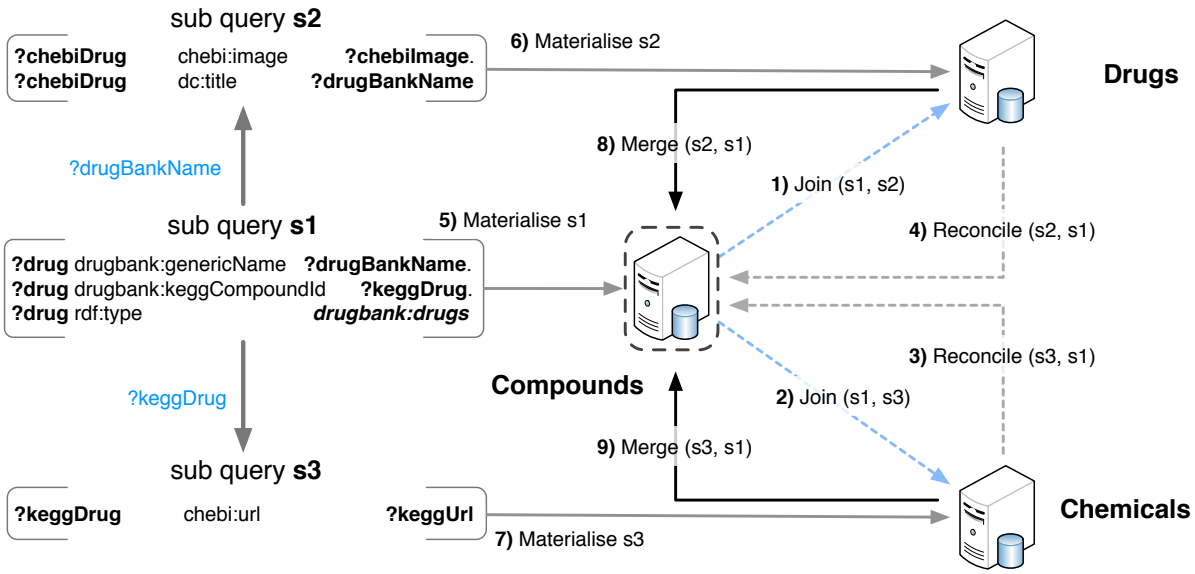


Fig. 6: Graphical illustration of the execution process for example query $Q_{ex.}$.

Algorithm 2 Plan Execution

Precondition: P a valid query plan, R_Q the AVALANCHE *Results Queue*

```
1:  procedure executePlan(P, R_Q)
2:      r ← ∅                                          ▷ r: the results
3:
4:      if isFederated(P) then                         ▷ P has more than 1 subqueries
5:          SortSubqueries(P)                          ▷ Sort subqueries in P by SEL^h_sq
6:          J_Q ← distJoin(P)                          ▷ distributed join of subqueries
7:          distReconciliation(J_Q)                    ▷ reconcile partial results
8:          S_Q ← distMaterialize(P)                   ▷ distributed materialisation (||)
9:          r ← distMerge(S_Q)                         ▷ merge partial results
10:     else
11:         r ← sparql(P)                              ▷ execute SPARQL query
12:     R_Q ← R_Q ∪ r                                  ▷ append results
```

et al., 2009] *bloom-filters* can be employed. Bloom-filters are space-efficient lossy bit-vector representations of sets by virtue of using multiple hash functions for recording each element. Finally results are materialised in parallel (line 8, steps 5,6,7 in figure) and then merged on the host corresponding to the first subquery – the one with the lowest estimated selectivity, (line 9, steps 8,9 in figure). To increase execution performance, since many plans contain the same or overlapping subqueries, a *memoization* strategy is employed. Hence, partial results are kept for the duration of the entire query execution and not just for the current plan. This acts as a site-level cache memory, bypassing the database altogether for "popular" result sets when resources permit.

When the merge is completed, the *Plan Executor* worker process will signal the AVALANCHE *Query monitor* via the *Results Queue*. Note that the finished plans do not contain the final results, as the matches are kept remotely. It is the *Query monitor's* re-

sponsibility to retrieve the results and update the overall state of the broker accordingly. In the remainder of this subsection we will describe in detail the inner-workings of the operations described above.

Distributed Join & Reconciliation The join and reconciliation procedures are detailed in Algorithms 3 and 4 respectively. Joining is implemented in a left-deep fashion while the reconciliation procedure is straight-forward.

Algorithm 3 Distributed Join

Precondition: P a valid query plan
Postcondition: J_Q a queue, containing the joins in order

```
1:  function distJoin(P)
2:      J_Q ← ∅                                          ▷ J_Q: joins queue
3:      𝕊 ← subqueries(P)                                ▷ 𝕊: set of all subqueries in P
4:      𝕊.sort()                                         ▷ sort by selectivity estimation SEL
5:      if isFederated(P) then                           ▷ P has more than 1 subqueries
6:          while 𝕊 ≠ ∅ do
7:              best ← 𝕊.pop()
8:              for sq ∈ 𝕊 do
9:                  best ⋈ sq                            ▷ remote join
10:                 J_Q ← J_Q ∪ {[best, sq]}             ▷ record join
11:     else
12:         sparqlRemote(P)                              ▷ Execute SPARQL but keep results remotely
13:     return J_Q
```

One important aspect to note is that the execution of a plan can be stopped (line 6 in Algorithm 4) if the cardinality of a join is 0. This information is recorded and fed back into the planner for dynamic pruning.

Algorithm 4 Distributed Reconciliation

Precondition: J_Q joins queue

```
1:  procedure distReconciliation(J_Q)
2:      J_Q.reverse()
3:      for [left, right] ∈ J_Q do
4:          κ ← reconcile(left, right)
5:          if κ = 0 then                                ▷ stop plan execution when cardinality = 0
6:              halt
```

Distributed Materialization & Merge The final execution phases are detailed in Algorithms 5 and 6 respectively. The materialization procedure is executed in parallel on all subquery hosts with the important note that locally kept selectivity estimations for each subquery in $\mathbb{S}_Q$are updated to actual join cardinalities, available at this stage remotely (line 5 in Algorithm 5). This information is later used to find out the host with

the highest partial result cardinality. This host (*best* in line 2 in Algorithm 6) is then used as the "hub" where all other partial results are merged (lines 3-5 in Algorithm 6).[21]

Algorithm 5 Distributed Materialization

Precondition: P a plan
Postcondition: $\mathbb{S}_Q$a queue containing the plan subqueries sorted by cardinality κ

```
1: function distMaterialize(P)
2:     S_Q ← ∅                                          ▷ S_Q: subqueries queue
3:     for sq ∈ P do
4:         κ ← materialise(sq)                          ▷ κ: the cardinality of partial results on sq
5:         S_Q ← S_Q ∪{[κ, sq]}
6:         if κ = 0 then                                ▷ stop plan execution when cardinality = 0
7:             halt
8:     S_Q.sort()                                       ▷ sort by κ
9:     return S_Q
```

Algorithm 6 Distributed Merge operation

Precondition: $\mathbb{S}_Q$ subqueries queue
Postcondition: r a valid SPARQL results set

```
1: function distMerge(S_Q)
2:     κ, best ← S_Q.popLeft()                          ▷ κ: the cardinality of partial results on best
3:     while S_Q ≠ ∅ do
4:         sq ← S_Q.popLeft()
5:         merge(best, sq)                              ▷ merge results from sq on best
6:     r ← getResults(best)                             ▷ retrieve the final results from best
7:     return r
```

4.6 Stopping the Query Execution

Since we have no control over distribution and availability of the RDF data and SPARQL endpoints, providing a complete answer to the query is an unreasonable assumption except for the cases involving few endpoints and rather simple queries. Instead, the *Query Monitor / Stopper* monitors for the following *stopping conditions*:

→ a global timeout set for the whole query execution,

→ returning the *first K* unique results to the caller,

→ to avoid waiting for the timeout when the number of results is $\ll$ K, we measure relative result-saturation. Specifically, we employ a sliding window to keep track of the last n received result sets. If the standard deviation (σ) of these sets falls below a given threshold, we stop execution. Specifically, we use Chebyshev's inequality: $1 - \frac{1}{\sigma^2}$ [Knuth, 1997].

[21] For brevity and graphical simplicity of Figure 6, the *Compounds* endpoint (in the middle) was also assigned to be the merge host.

All of the above mentioned stopping conditions can be enabled / disabled independently and in any combination required by a given use-case or desired by the user.

5 Evaluating Avalanche's Robustness Against Messiness

In the introduction we claimed that the Avalanche system provides the capability to query the messy Web-of-Data. Specifically, we claimed that the proposed system: (1) *makes no assumptions about data distribution, schema, availability, or partitioning* and is skew resistant for some classes of queries, (2) provides *up-to-date results* from distributed indexed endpoints, (3) is *adaptive* during execution adjusting dynamically to external network changes, (4) *does not require detailed fine-grained ex-ante statistics* with the query engine, and (5) is *flexible* as it makes limited assumptions about the structure of participating triple stores.

Avalanche is able to provide up-to date results without any ex-ante statistics (2 and 4) by accessing participating triple-stores at run-time and is open due to the limited assumptions it makes on triple-stores (5). Whilst skew resistance (1) and adaptiveness (3) seem possible due to its multi-plan competitive planing/execution strategies (see Sections 4.2 and 4.5) it has not been shown that these strategies are actually successful.

In the following we describe the experimental evaluation of the Avalanche system. Specifically, we will provide empirical evidence ascertaining Avalanche's planner quality and the system's overall robustness to varying data distributions and network conditions such as different latencies and endpoint unreliability. Specifically, we evaluate Avalanche's planer quality as well as robustness against network latency and endpoint stability (in Section 5.1) using a real world dataset. In addition, we show Avalanche's robustness against various data distributions (Section 5.2) using a synthetic dataset.

Experimental setup For all experiments a cluster of 6 physical machines with 64GB of RAM, 24 AMD Opteron 6174 Cores @2.2 GHz, and running Debian GNU/Linux 6.0.6 64bit was used, connected by a 1 gigabit ethernet switch. In addition the Avalanche broker was executed on a separate machine with 72GB of RAM, 8 Intel(R) Xeon(R) CPU X5570 Cores @2.93GHz, and running Fedora release 12 (Constantine) 64bit. For all evaluations the following stopping conditions were considered unless specified otherwise:

→ a global *timeout* of 300 seconds (5 minutes),
→ *first K* unique results set to 1000 and
→ *relative-saturation* of 90%.

Additionally, the concurrency limit was set to 128 concurrently executing plans.

5.1 Evaluation Setting I: Analysing Avalanche with real-world data

To evaluate the generalisability of our results to a real-world setting we chose a real-world dataset specifically tailored for the evaluation of federated RDF stores. This subsection first outlines the dataset, its distribution to hosts, the queries used and then discusses Avalanche's execution results on this dataset.

The Data and its Distribution We chose the recently published Fedbench[22] [Schmidt et al., 2011] dataset as it comes pre-partitioned using a real-world partitioning schema and, additionally, offers 36 SPARQL queries. For summarised statistics about each participating dataset refer to Table 1.

Table 1: Fedbench datasets statistics

Collection	Dataset	version	# triples
Cross Domain	DBpedia subset	3.5.1	43.6M
	Jamendo	2010-11-25	1.05M
	NY Times	2010-01-13	335k
	GeoNames	2010-10-06	108M
	LinkedMDB	2010-01-19	6.15M
	SW Dog Food	2010-11-25	104k
Life Sciences	DBpedia subset	3.5.1	43.6M
	Drugbank	2010-11-25	767k
	KEGG	2010-11-25	1.09M
	ChEBI	2010-11-25	7.33M
SP²Bench	*SP²Bench* 10M	v1.01	10M

[a] Data available from http://fedbench.fluidops.net/resource/Datasets

Following the natural partitioning of the benchmark we adopted the assumption that each dataset is published on its own distinct server. For bigger datasets such as Geonames and DBPedia we assumed in addition that the publishers decided to further split the data into multiple RDF stores. We captured this by splitting some of the larger datasets as detailed in Table 2. Hence, additional distribution messiness was introduced by splitting the Geonames triples randomly over 11 hosts while for DBpedia larger dumps were distributed to single hosts and the smaller ones were integrated into the *Other* AVALANCHE endpoint.

The Queries The triple store[23] we used for implementing AVALANCHE endpoints does not currently support SPARQL features beyond traditional BGP pattern matching. Hence, we ignored all Fedbench queries that contain the OPTIONAL and FILTER graph pattern modifiers. This is a limitation of the current system and evaluation, which we discuss in detail in Section 6. Additionally, as UNION graph patterns are not supported either, queries containing the operator were split and executed as separate queries, which is aligned with the common practice of executing unions as individual subqueries in parallel. We supplemented the resulting 33 Fedbench queries with another 5 more complex queries from the life sciences domain, as listed in D. The translation table to the original names (where applicable) is available in B.

[22] http://code.google.com/p/fbench

[23] An in-house and update-able extension of *Hexastore* was used as the RDF store technology behind all AVALANCHE endpoints in our evaluations.

Table 2: The distribution of the Fedbench dataset to AVALANCHE hosts

Dataset	AVALANCHE Host	#triples
NY Times	News	314k
LinkedMDB	Movies	6.14M
Jamendo	Music	1.04M
SW Dog Food	SW	84k
KEGG	Chemicals	10.9M
ChEBI	Compounds	4.77M
Drugbank	Drugs	517k
SP2B-10M	Bibliographic	10M
Geonames	Geography_i, where $i \in [1, 10]$	$\approx$ 9.9M
	Geography_11	7.98M
DBPedia subset	Infobox_Types	5.49M
	Infobox_Properties	10.80M
	Titles	7.33M
	Articles_Categories	10.91M
	Images	3.88M
	SKOS_Categories	2.24M
	Other	2.45M

Experiment #1: AVALANCHE vs. a Baseline System In this first experiment we intend to better understand through empirical evidence, *the performance gains (or potential shortcomings) that the computational model embraced by* AVALANCHE *introduces*. Hence, we implemented a *baseline* prototype where the core idea of concurrently executing multiple *simpler* decompositions of the original query is dropped. In contrast to AVALANCHE the query answer-set is constructed by:

- keeping relevant state (i.e., partial results) in a local repository and
- executing a single optimal query plan generated akin to traditional query optimisation techniques.

Although there are multiple possible execution models that could be considered baselines, one approach is to first multicast query Q to all participating sites. Second, each site would remove triple patterns for which it has no match from Q and return the matching triples. Third, Q would be run against a local repository of the triples returned from all participating hosts. The decision to discard triple patterns—in effect mapping $Q \mapsto Q_{known}$, where Q_{known} is the part of the query known to the server—is carried out by each participating endpoint individually and is implemented as defined in Equation 8:

$$T_{Q_{known},h} = \{tp_i \mid \forall tp_i \in T_{Q,h}, \text{iff } card(tp_i, h) > 0\} \tag{8}$$

where $T_{Q_{known},h}$ represents the "known" set of triple patterns composing query Q_{known} on the current host h. Other triple pattern exclusion rules can be imagined, i.e. discard all triple patterns if the predicate belongs to an unknown namespace – provided namespace

information is available. After all or some of the partial results are retrieved from the remote SPARQL endpoints, they are stored in a local RDF store. Since in the case of SPARQL SELECT queries the answer-sets R_i are tables where columns correspond to projection variables and therefore not graphs G_i as would be the case of SPARQL CONSTRUCT queries, a translation process from tuples to triples needs to be implemented. This is a necessary step as to reconstruct locally the subgraph $G_{known} = \bigcup G_i$. A solution would be to transform each of the Q_{known} SELECT queries to equivalent CONSTRUCT queries. Finally, the engine is left with the task of re-executing the original query Q on the local graph G_{known}.

Limitations of the Baseline System While conceptually simpler, a number of hurdles render the implementation non-trivial. First, it is possible that some of the reduced queries Q_{known} may not contain any selective triple patterns from Q because the respective hosts do not "understand" those patterns. In the worst case the reduced $Q_{known} \equiv < s, p, o >$ which would trigger the requester to retrieve the entire remote knowledge-base. Second, since the final results for Q can only be computed after obtaining G_{known} two execution strategies emerge:

i) Wait until all G_i partial graphs are retrieved and then execute Q on G_{known}. This is suitable for cases where the partial graphs are inexpensively obtained and/or the query is complex.

ii) Build the final result-set incrementally by executing Q every time a partial graph G_i is merged with the local G_{known} repository. This strategy obviously pays off when (some) partial triples sets are expensive to obtain additionally offering the possibility of an early stop when Q is satisfied without having to wait for all partial graphs. However, it incurs the cost of executing Q with each retrieved partial set of RDF triples i.e., returned by each site.

```
1 PREFIX ex: <http://example.org/>
2 SELECT * WHERE {
3     ?x ex:p1 ?y .
4     ?y ex:p2 ?z .
5     ?z ex:p3 ?u .
6 }
```

Listing 6.3: Example query Q'_{ex}.

Finally, the method is not complete since it is possible that $\bigcup G_i \subset G_{needed}$, where G_{needed} is the minimal set of triples needed to construct the complete result set for Q. For example consider the case illustrated in Figure 7 where query Q'_{ex} (from Listing 6.3) executes over two sites. By this strategy Q'_{ex} produces no results even though the complete result-set contains two tuples. In contrast AVALANCHE is (eventually) complete since it considers all possible decompositions of Q and not just some decompositions like Q_{known}.

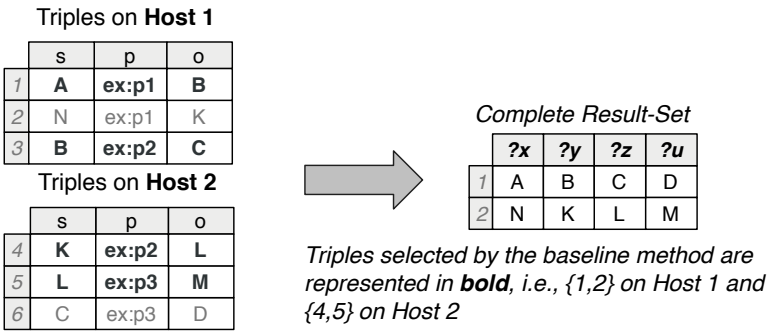


Fig. 7: Triples distribution for two hypothetical sites with the complete result-set for query Q'_{ex}.

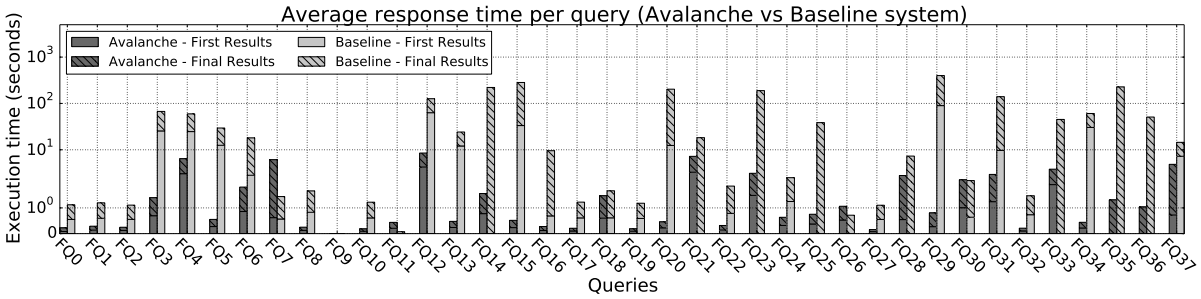


Fig. 8: Average query execution times for each of the Fedbench queries. AVALANCHE vs. the Baseline System.

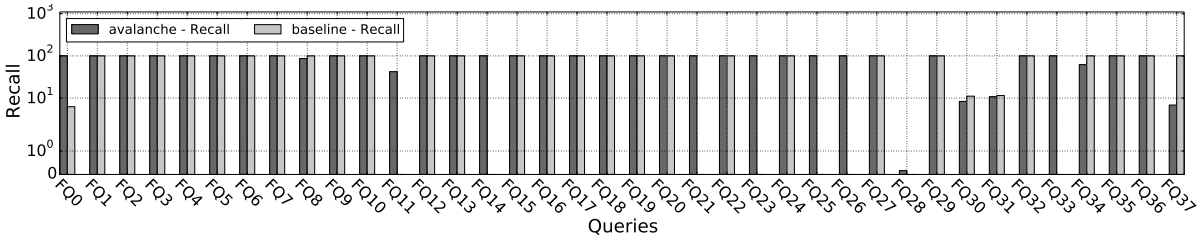


Fig. 9: Recall for each of the Fedbench queries. AVALANCHE vs. the Baseline System.

Results Based on the assumption that the selectivity distribution of the generated Q_{known} subqueries on participating endpoints is ZIPF-ian, we chose to implement the pipelined execution model due to its obvious performance benefits. Furthermore, the same asynchronous execution paradigm as in AVALANCHE was used in the baseline, while G_{known} was implemented by a fast in memory indexed RDF store[24]. A consequence of this choice is that the same stopping conditions that AVALANCHE employs can be used to determine wether the engine should stop the query execution or not, hence, eliminating other unknown hidden factors when comparing the two systems.

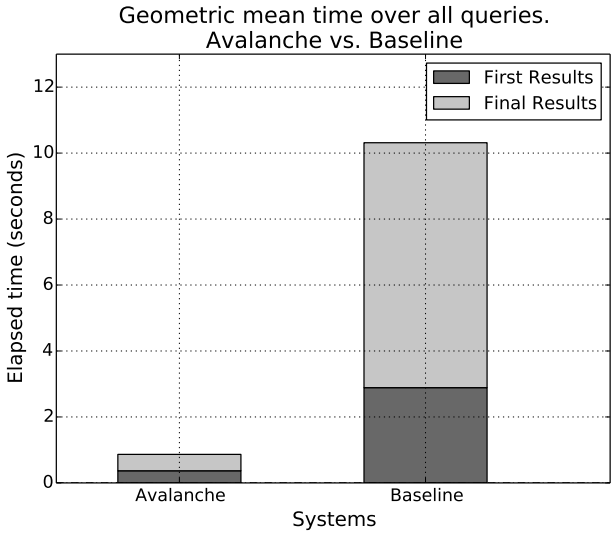


Fig. 10: Geometric mean of the execution time over all queries: AVALANCHE vs. the Baseline System.

The time taken to complete all the considered Fedbench queries by both systems is graphed in Figure 8. With very few exceptions AVALANCHE proved to be faster than the *baseline* system. When retrieving first results the baseline system is slower than AVALANCHE in 65% of the queries, becoming slower for 92% of the queries by the time final results are retrieved. This is better captured in Figure 10, where the geometric mean over all queries is computed. Clearly, for the 38 selected Fedbench queries AVALANCHE exhibits superior average performance for both cases: retrieving first results and achieving query completion.

Furthermore, as mentioned previously the baseline system is not guaranteed to be complete, a fact exhibited by queries: $FQ11$, $FQ14$, $FQ21$, $FQ23$, $FQ25$, $FQ26$, $FQ28$ and $FQ33$ as seen in Figure 9, which depicts the recall for all queries. In contrast, AVALANCHE exhibits full recall for most queries with the exception of queries: $FQ8$, $FQ11$, $FQ28$, $FQ30$, $FQ31$, $FQ34$ and $FQ37$ under a time-out of 5 minutes (the same was set for the baseline). The *ground-truth* —total number of results— used to compute the recall was obtained by running all AVALANCHE plans exhaustively acquiring thus all

[24] We used the IOMemory RDF store provided by the `rdflib` package: https://github.com/RDFLib

Table 3: Statistical information and query runtime breakdown for the Baseline system on all queries

Query	# query runs[a]	Average load time (s)[b]	Average query time (s)[c]	Total triples received[d]
FQ0	3	0.0003	0.1748	1
FQ1	1	0.0065	0.5123	25
FQ2	2	0.0004	0.2766	1
FQ3	18	0.9735	1.0130	206211
FQ4	17	1.5448	0.0372	385055
FQ5	15	0.8346	0.0414	132931
FQ6	6	1.4098	0.9325	79857
FQ7	3	0.0961	0.1853	2810
FQ8	1	0.1052	0.5886	1158
FQ9	0	-	-	-
FQ10	1	0.0303	0.5261	318
FQ11	0	-	-	-
FQ12	19	2.3080	0.3670	595434
FQ13	2	3.3561	0.2621	138132
FQ14	4	28.7477	0.2759	983324
FQ15	6	21.8694	11.1893	1114704
FQ16	3	1.9255	0.1989	54619
FQ17	1	0.0302	0.5258	554
FQ18	3	0.1056	0.2032	3816
FQ19	1	0.0123	0.5226	249
FQ20	6	18.0261	0.0934	947537
FQ21	8	1.8014	0.0698	153450
FQ22	2	0.1137	0.3232	2430
FQ23	15	7.4315	0.0527	976104
FQ24	3	0.1865	0.1760	6842
FQ25	18	1.6552	0.0344	284896
FQ26	1	0.0870	0.5571	1139
FQ27	1	0.0007	0.5519	2
FQ28	2	2.3257	0.2940	44087
FQ29	5	34.8218	2.7780	1705932
FQ30	2	0.3084	0.3861	9472
FQ31	2	26.7686	13.4666	776692
FQ32	1	0.0517	0.5352	655
FQ33	18	1.5853	0.4695	288054
FQ34	1	0.7198	28.2120	19367
FQ35	5	24.8012	0.1232	1114611
FQ36	18	1.7777	0.4739	386434
FQ37	1	3.2252	1.3490	87599

[a] the input query is run repeatedly every time new triples are received
[b] average time – in seconds – to load the newly received triples into the local RDF store
[c] average time – in seconds – taken for each input query run
[d] total number of triples transferred over the network from all endpoints

possible results for each query. This was achieved by disabling all the stopping conditions: timeout, first-k results and relative saturation.

The baseline system although slower for most benchmark queries and incomplete for some, exhibits some positive properties. First, it is of a much more simple design than Avalanche and finally for some classes of queries it can be faster than Avalanche.

For example for query $FQ7$ the baseline system completes with 4.6 seconds faster than AVALANCHE while for query $FQ30$ first results are retrieved marginally (0.37 seconds) faster than AVALANCHE. As stated above one of the main design limitations of the baseline is represented by the fact that completeness cannot be guaranteed. Even though we implemented the baseline using the same concurrent asynchronous query execution paradigm as in AVALANCHE a number of potential bottlenecks still exist. A first limiting factor is the way in which the query is being executed: by fetching all pertinent (according to Equation 8) triples locally. Intuitively, at least for more demanding classes of queries (i.e., with more joins, or complex shapes), this can easily lead to a large portion of triples to be identified as "pertinent" for the given query and therefore transferred locally. Looking at Table 3 we can clearly observe that for 50% of the benchmark queries, the baseline retrieves anywhere between 100'000 to 1'700'000 triples, while for very few queries the number of triples retrieved counts in the hundreds. Clearly, this represents a bottleneck since not all triples are received at the same time, and in some cases those triples that contribute to the final result are found later in the execution of the given query. Another potential bottleneck is represented by the local RDF store we employed. We opted for an in-memory indexed store to diminish the performance penalties introduced by the loading of new triples as they arrive and at the same time offer high performance for most queries. As can be observed in Table 3 most queries are answered on average below 1 second, however for some queries (e.g., $FQ15$, $FQ29$, $FQ31$ and $FQ34$) the time to rerun the original query on local data is on average quite high ranging from ca. 3 seconds to ca. 30 seconds. This could be explained by the set-based join algorithm used (more expensive than sorted merge-join) since the RDF store does not keep sorted indexes (but dictionary based) to aid the loading / indexing process at the expense of slower execution times for more complex queries.

In light of these results, we can safely say that AVALANCHE exhibits significant performance and conceptual benefits over the naive baseline system.

Experiment #2: Planner Quality Assessment In this second experiment we intend to analyse the *quality* of the planning algorithm and cost model that AVALANCHE uses. Consequently, we:

- compare the performance exhibited by AVALANCHE with that of a similar system driven by an *oracle planner* and,
- observe the relative ranking of productive plans within the query plan universe P_Q as generated by the AVALANCHE plan generator.

Comparison to an Oracle Planner In order to observe to what extent the asynchronous concurrent execution of plans improves the overall performance of query answering in AVALANCHE we constructed an *oracle planner* (see Definition 5).

Definition 5. *An **oracle planner** is a plan generator connected to an oracle, akin to an **oracle machine**, i.e. a Turing machine connected to an oracle.*

Table 4: Total possible plans and first productive plan rank as generated by AVALANCHE

query	*FQ0*	*FQ1*	*FQ2*	*FQ3*	*FQ4*	*FQ5*
max plans[b]	26^1	26^2	26^3	26^5	26^5	26^4
# plans[c]	6	26	1	18	324	18
# productive plans[d]	6	1	1	1	1	1
1^{st} plan	**1**	**25**	**1**	**1**	**2**	**3**
query	*FQ6*	*FQ7*	*FQ8*	*FQ9*	*FQ10*	*FQ11*
max plans[b]	26^4	26^4	26^1	26^1	26^1	26^2
# plans[c]	180	2592	1	0	1	26
# productive plans[d]	1	2	1	0	1	1
1^{st} plan	**23**	**48**	**1**	-[a]	**1**	**2**
query	*FQ12*	*FQ13*	*FQ14*	*FQ15*	*FQ16*	*FQ17*
max plans[b]	26^5	26^7	26^6	5^{26}	26^3	26^3
# plans[c]	18	1	10	10	7	1
# productive plans[d]	1	1	1	1	1	1
1^{st} plan	**6**	**1**	**6**	**2**	**7**	**1**
query	*FQ18*	*FQ19*	*FQ20*	*FQ21*	*FQ22*	*FQ23*
max plans[b]	26^4	26^5	26^3	26^5	26^2	26^5
# plans[c]	126	1	5	594	24	180
# productive plans[d]	1	1	1	1	1	1
1^{st} plan	**20**	**1**	**1**	**71**	**9**	**1**
query	*FQ24*	*FQ25*	*FQ26*	*FQ27*	*FQ28*	*FQ29*
max plans[b]	26^3	26^3	26^5	26^3	26^9	26^5
# plans[c]	1	18	49	1	270	45
# productive plans[d]	1	1	1	1	1	1
1^{st} plan	**1**	**2**	**1**	**1**	**1**	**1**
query	*FQ30*	*FQ31*	*FQ32*	*FQ33*	*FQ34*	*FQ35*
max plans[b]	26^2	26^2	26^1	26^{16}	26^{12}	26^{16}
# plans[c]	104	104	1	18	1	10
# productive plans[d]	1	3	1	1	1	0
1^{st} plan	**27**	**20**	**1**	**6**	**1**	-[a]
query	*FQ36*	*FQ37*				
max plans[b]	26^{11}	26^6				
# plans[c]	18	324				
# productive plans[d]	0	1				
1^{st} plan	-[a]	**17**				

[a] query has no results
[b] maximum number of plans if no triple-pattern cardinalities are available ≡ upper bound
[c] maximum number of possible plans deduced when triple pattern cardinalities are considered
[d] total number of plans (from all possible plans) for which results are found

A drop-in replacement for the AVALANCHE *Plan Generator*, the oracle planner has perfect knowledge about which of the AVALANCHE generated plans are productive (i.e., have results) and which are not (i.e., do not find any results). To obtain the productive plans for each query, we serialised the plans for which results were found while running AVALANCHE without stopping conditions, to disk. We then order these plans according to the same order as AVALANCHE. Consequently, the oracle planer only generates the plans for which results are found without the time penalty incurred by the exhaustive plan space traversal of the cost-based *Plan Generator*.

It is important to note that for most queries with the exception of $FQ0$, $FQ7$ and $FQ31$ (see Table 4) there is only a single plan which is productive and therefore the *oracle planner* is in this cases equivalent with an omniscient planner where the optimal query decomposition is found.

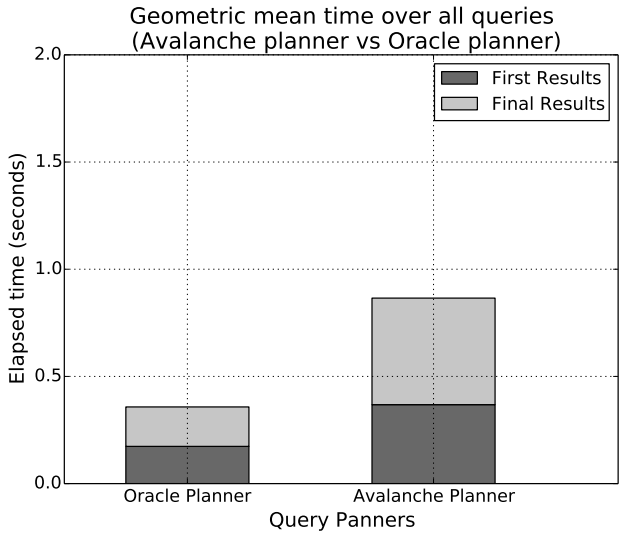


Fig. 11: Geometric mean of the execution time over all queries: Oracle vs. AVALANCHE Planner.

For the experiment we ran all benchmark queries with the oracle planner and compared the performance of query execution to the AVALANCHE cost model based planner. The number of productive plans for all of the benchmark queries is reported in Table 4. As can be seen, all queries feature 1 productive plan (or 0 if query has no results) with the exception of queries $FQ0, FQ7$ and $FQ31$ which produce 6, 2 respectively 3 productive plans.

The results of running all 38 Fedbench queries comparing the standard AVALANCHE planner with the *oracle planner* are depicted in Figure 12, while the geometric mean over all queries when comparing the execution times yielded by the two planners is shown in Figure 11. While for 25 of the queries the absolute elapsed time (wall-clock time) difference is negligible as seen in Figure 12, for queries $FQ1$, $FQ4$, $FQ6-7$, $FQ18$, $FQ21$, $FQ23$, $FQ28-31$, $FQ33$ and $FQ37$ AVALANCHE was be-

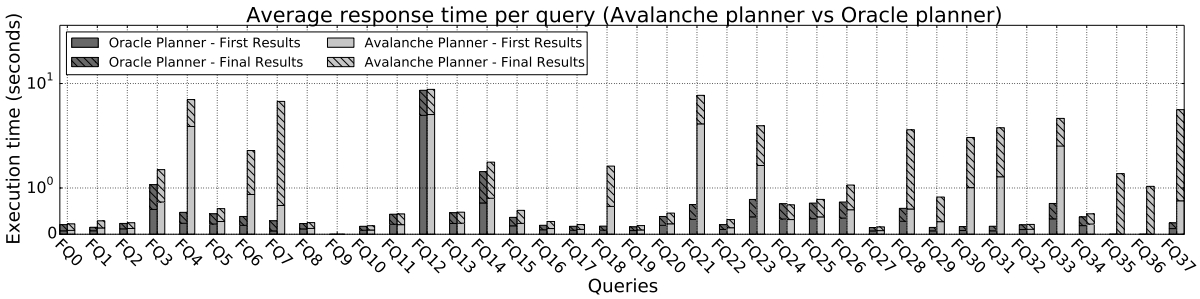


Fig. 12: Average query execution times for each of the Fedbench queries. AVALANCHE planner vs. Oracle planner.

tween ≈ 2 to ≈ 33 times slower than the oracle approach. However, looking at Figure 11, AVALANCHE was ≈ 2.5 times slower in the geometric mean than the oracle driven system over all benchmark queries.

In general this difference is to be expected. The effort of discarding (and executing) unproductive plans in conjunction with the plan space exploration takes time. Hence, the AVALANCHE planner is naturally slower than a no-effort planner (like the oracle planner). However, as exhibited by Figure 12 the delays are clearly limited and acceptable to many applications. Hence, AVALANCHE exhibits a good performance in the conditions of this evaluation when acting solely on join-estimate heuristics.

Plan ranking As can be seen in absolute values in Table 4 and normalised relative to total number of plans in Figure 13 AVALANCHE succeeds in assigning a low rank (1 ≡ best rank) to the first productive plan. When the number of possible plans is large, the simple selectivity-estimation-based cost model will assign higher ranks, as is the case of query $FQ21$ where the first productive plan is the 71^{st} plan generated out of 594 possibilities. However, due to the asynchronous-concurrent manner in which plans are executed, the negative effect of assigning higher ranks to plans (the rank is equivalent to the plan's generation order) is mitigated to a relatively high degree as shown in the previous analysis agains the oracle planner, i.e. non-productive plans are quickly discarded after the first empty join.

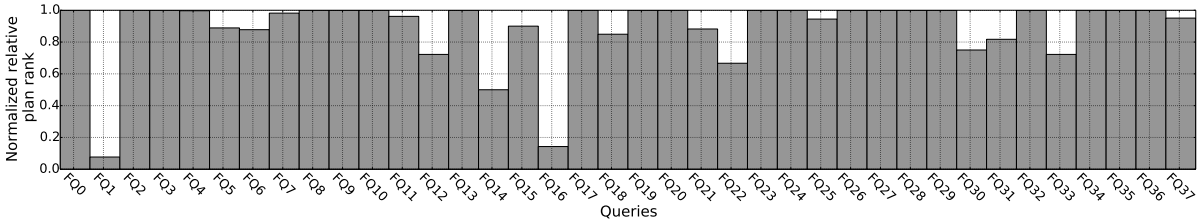


Fig. 13: Normalised relative plan ranking: first plan compared to the possible number of plans / query for each Fedbench queries.The higher the bar the better, i.e. productive plans get executed sooner.

Experiment #3: Varying Network Latency Changing network conditions can impede the execution of any distributed SPARQL processing. Two critical network factors stand out: bandwidth and latency. Since the slowdown effect of a low-bandwidth connection can in general be overcome with a certain degree of success by either compressing the message or making use of binary communication protocols and since AVALANCHE employs *bloom filter* optimised joins to reduce communication I/O, we decided to focus our attention in this experiment on connection latency. The majority of requests in the AVALANCHE system are between the AVALANCHE broker and the participating endpoints. Hence, for this experiment the connection between the broker and each endpoint was routed through a TCP delayer proxy, which would introduce delays according to a predefined configuration. We chose to simulate three types of latency distributions:

- *No Delay* → a *local cluster* network with negligible (close to 0 s) connection latency,
- *Gamma 1* → a *fast network* with an average connection latency of 0.3 seconds. Simulated by a gamma distribution with $\alpha = 1$ & $\beta = 0.3$ (Figure 14),
- *Gamma 2* → a *slow network* with an average connection latency of 3 seconds. Simulated by a gamma distribution with $\alpha = 3$ & $\beta = 1.0$ (Figure 14).

Additionally, the TCP socket buffer size was set to the standard value of 16KB.

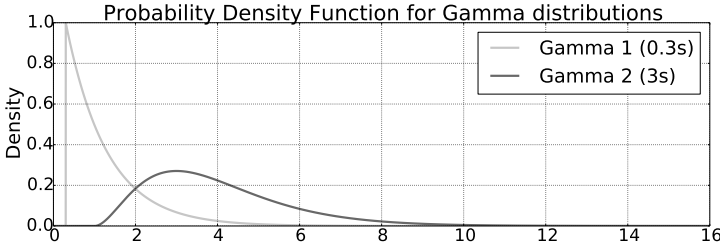


Fig. 14: Probability density function (pdf) for the simulated *Gamma* 1 and *Gamma* 2 latency distributions.

AVALANCHE successfully finds results for all the considered benchmark queries under all simulated latency variations. Looking at Figure 15 we can clearly observe that the speed with which AVALANCHE answers queries across the different connection types increases dramatically as we move towards slower connections like *Gamma 2*. First, AVALANCHE retrieves query specific statistics (e.g., triple pattern cardinalities and total triples) from participating endpoints. For the 0 latency setup *No Delay* this phase completes on average in 0.05 seconds and is therefore negligible compared to the overall query execution time. For the slower networks *Gamma 1* and *Gamma 2* the statistics gathering phase takes on average 1.22 seconds and 7.54 seconds respectively.

Although these execution times are significantly higher they are mainly dominated by the network connection latency when optimised remote endpoints are employed. This fact can be observed from the low response time for the same statistical information when network latency is 0. Next, results are produced after an average of 0.36 seconds when connection latency is negligible, while for the *Gamma 1* and *Gamma 2* cases first results are found after an average of 2.93 seconds and 20.64 seconds respectively. The situation

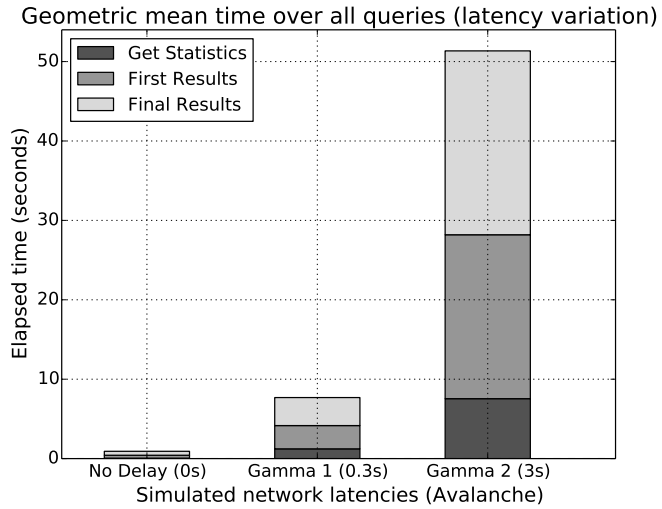


Fig. 15: Geometric mean of the execution time over all queries for the three connection setups.

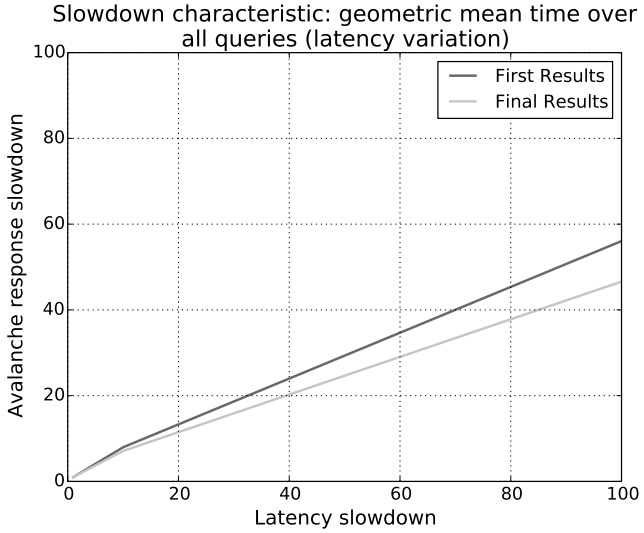


Fig. 16: Slowdown introduced by the three connection setups.

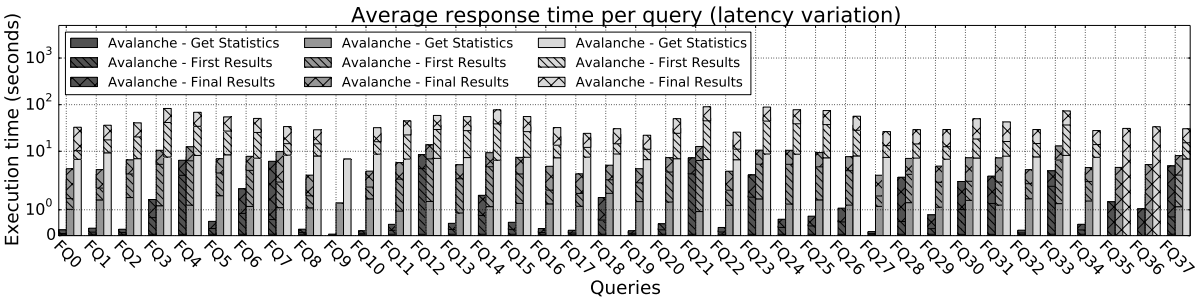


Fig. 17: Average response time for each Fedbench query under different latency distributions. The graph differentiates between the time necessary to get the statistics, execute the first plan, and execute all plans.

is similar for achieving the stop condition or final results: 0.49 seconds on average for the *No Delay* setup, 3.52 and 23.15 seconds on average for the *Gamma 1* respectively *Gamma 2* setups. Although this performance decrease is dramatic, AVALANCHE exhibits a sub-linear slowdown as graphed in Figure 16 compared to the broker-endpoints average latency slowdown.

This behaviour is attributed to AVALANCHE mainly because of its adaptive asynchronous design. In essence plans that return quickly are favoured by the asynchronous scheduling *Results Queue*. As a consequence, AVALANCHE is largely dependent on the *critical plan* for first results. The *critical plan* should ideally be the first productive plan. However, given that network conditions are uncontrollable, a slower plan might produce results faster because it shares a faster network connection. This is also observed in Figure 17, where the individual average times for answering all Fedbench queries $FQ_i, i \in [0, 37]$ queries under all three network conditions are graphed. As the broker-endpoints connections experience more lag, AVALANCHE exhibits a stable behaviour overall depending mainly on the *critical plan(s)*, albeit slower with the slowdown depicted in Figure 16.

Experiment #4: Varying Endpoint Availability Another source of messiness stems from the uncontrollable nature of the underlying communication protocol stacks on the Web as well hardware and physical crashes of servers and routers. There is no guarantee that a host replying to requests at any given moment T will be available at time $T + \Delta t$. To observe the behaviour of AVALANCHE in such a case we have designed an experiment, where some hosts disappear during query execution.

First, in order to have multiple plans per query we replicated some of the Fedbench endpoints used throughout this experimental setup. Specifically, we replicated the following AVALANCHE endpoints with a factor of 2: the *News, Movies* and *Music* in the Cross Domain collection and *Drugs* in the Life Sciences collection (see Table 2). This resulted in the increase of total number of triples over all hosts by about 8 million additional assertions. Furthermore, the already burdened physical machines had to support the 4 additional replicated endpoints.

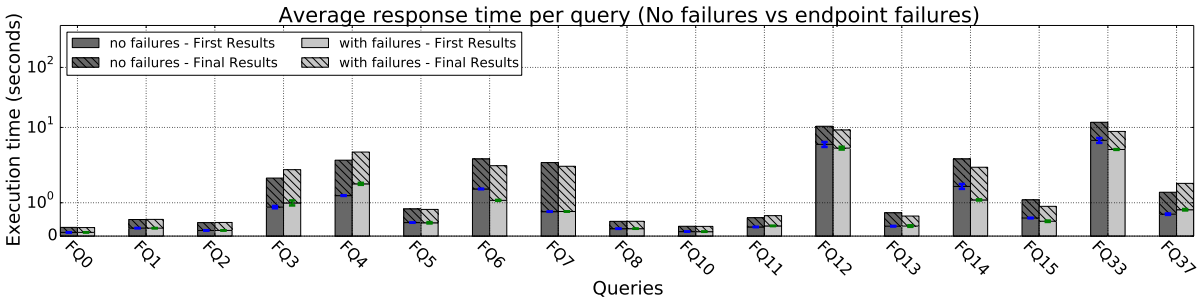


Fig. 18: Average response time for Cross Domain and Life Sciences Fedbench queries when endpoints fail.

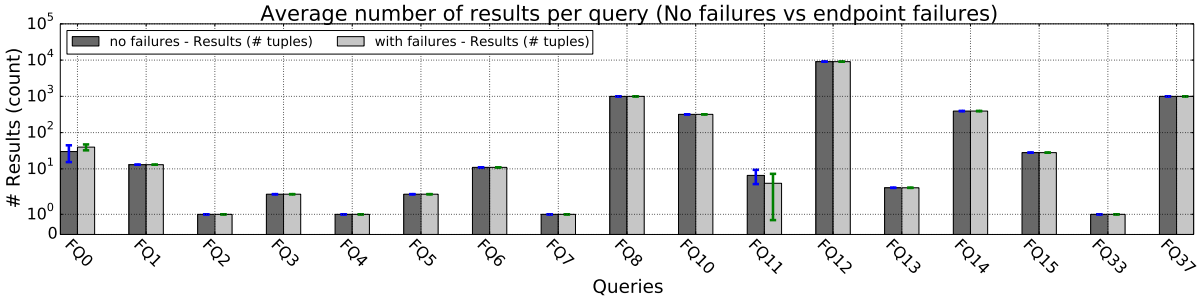


Fig. 19: Average # of results time for Cross Domain and Life Sciences Fedbench queries when endpoints fail.

Then, to emulate a crash the replicated endpoints were started in a "fail" mode, meaning that they would abruptly terminate themselves immediately after reporting the triple pattern cardinalities. This case is most interesting as the hosts will be considered by the *Query Planner* component as it received cardinalities from them, even-though all query plans containing subqueries allocated to them will fail to execute. The two other cases—the host being unavailable during either the source selection or statistics gathering phase—are less interesting as they are handled by design (i.e., the hosts are not even considered in the planning). We compared AVALANCHE when replicated hosts would fail seamlessly during query execution with the case when the replicas would not fail. Note that the obtained results should not be directly compared to results obtained elsewhere in this section, as the AVALANCHE endpoints were simulated on some of the physical nodes, which experience additional load in this replicated setting.

Figure 18 graphs the arrival time of the first and total results for the cross domain and life sciences queries ($FQ_i, i \in [0, 15] \cup [33, 37]$) and Figure 19 graphs the average number of results obtained over the same queries. Note that queries $FQ9$, $FQ35$, and $FQ36$ were not considered since they produce no results be default, while query $FQ34$ could not be run in the fully replicated scenario since the physical machine did not have enough resources to accommodate the extra replicated servers in this case.
AVALANCHE's *Plan Generator* adapts dynamically to external changing conditions, such as endpoints going offline, due to various reasons. Such events are usually detected when

a plan that contains at least one subquery assigned to an offline host is executed. Upon detection, the planner's internal state is dynamically readjusted first by removing the corresponding row for the host from the *Plan Matrix* P_M and secondly by pruning all partial plans containing the offline host generated up to the detection moment. In most cases AVALANCHE is not impacted by the fact that a host has failed when at least another alternate plan to produce results exists. Of course, if all query relevant hosts fail, then the query will timeout without any results found. As the results indicate AVALANCHE is able to return a result set of similar size as the one without disappearing hosts within a similar time-frame as in the stable host setting.

5.2 Evaluation Setting II: Analysing AVALANCHE with synthetic data

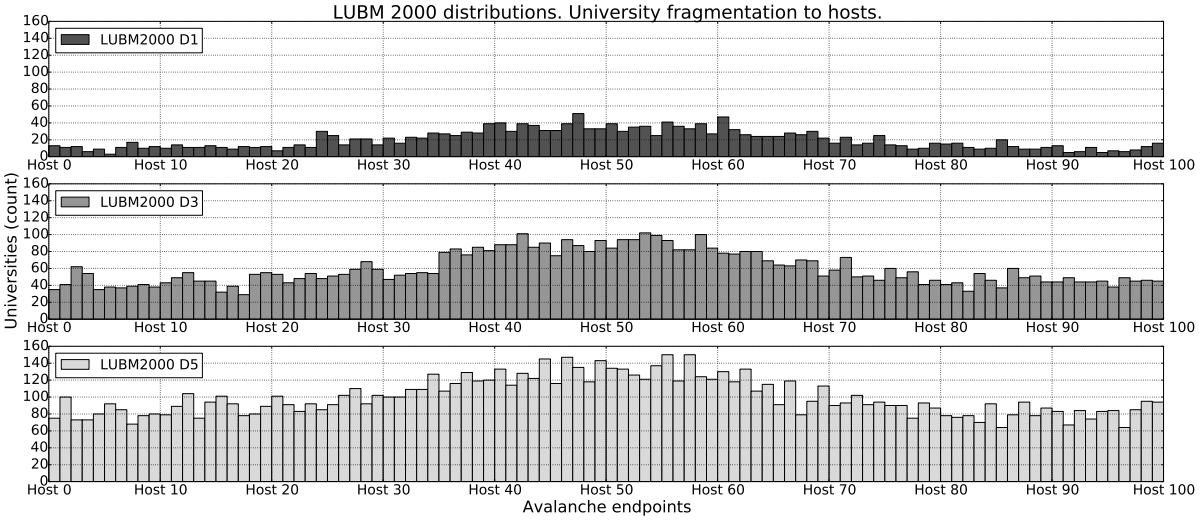


Fig. 20: The data distributions chosen over 100 Hosts. The y-axis denotes the number of universities about which a host contains information.

One of the key characteristics of the WoD is represented by its semantic heterogeneity stemming from a plethora of intertwining applications domains. Currently this aspect alone represents an important part of a federated query's selectivity. However, it is not inconceivable that in the future *schema-homogeneous* partitions of the WoD will increase in size reducing the usefulness of schema/vocabulary information during planning. These kind of *instance-level messy* distributed RDF datasets, hence, significantly complicates distributed query processing as it is unclear if triples matching one triple pattern from one host are likely to join with matches to a second triple pattern from the same host or another. This kind of messiness attenuates the effect of locality.[25] While AVALANCHE was not designed with the intend of addressing *instance-level messiness* we investigate the behaviour of our proposed execution paradigm when individual instances (triples) are

[25] Note that supporting this messiness is one underlying principles of the Semantic Web, as everyone can annotate any resource with some triple.

spread across a large number of *semantically-homogenous* hosts with increasing degrees of messiness.

To this end we used the synthetic LUBM benchmark dataset [Guo et al., 2005]. Specifically, we generated the LUBM2000 benchmark configuration, resulting in 2000 universities, and accounting to a total of 276 million triples. In contrast to the previous setup, where 26 **schema-heterogeneous** endpoints were used, a total of 100 **schema-homogeneous** endpoints are created. Such a setup allows us to flexibly mimic instance-level "distribution messiness" by reassigning triples to hosts. Note, that this setup situates AVALANCHE in a ***worst case*** scenario, where the *Source Discovery Phase* reports a large number of semantically-identical sources—all sharing the same schema—but with an unknown distribution of triples.

The Data and its distribution As illustrated in Figure 20, the LUBM triples were allocated to hosts according to the three *LUBM2000 D1*, *LUBM2000 D3*, and *LUBM2000 D5* distributions (in short $D1$, $D3$ respectively $D5$). The degree of distribution messiness increases with each case as detailed in the remainder of this section.

A *coarse-grained level of messiness* is achieved in the *LUBM2000 D1* data-distribution. Here all data belonging to a university is placed on the same host. To simulate various levels of server load we assign universities to hosts using the following procedure. Half the universities are randomly assigned to a host ensuring a basic load for each host. The second half of the universities are assigned to a host by drawing the host id from a normal distribution with mean $\mu = 50$ and standard deviation $\sigma = 14$. This leads to a higher load for some hosts (towards the middle of Figure 20).

To achieve a higher degree of instance-level messiness *LUBM2000 D3 & LUBM2000 D5* additionally distribute triples of one university across 3 or even 5 hosts. The initial host for a university is still determined using the same procedure as with $D1$. Once that host is determined, however, 2 (or 4) additional hosts are randomly selected. For $D3$ each university's triples are distributed over 3 hosts using a normal distribution with $\mu = 1.5$ and $\sigma = 0.3$. similarly, for data distribution $D5$, each university's triples are distributed over 5 hosts using a normal distribution with $\mu = 2.5$ and $\sigma = 0.5$. Hence, the bulk of the university's data is still on one host with part data distributed elsewhere. This mimics a Brownian motion of the data away from its originating source – one host contains most of the data while the rest is diffused to other hosts with the chosen probability density function. Consequently, as Figure 20 shows, the hosts will have data about more universities.

The Queries Although we employed the LUBM benchmark data generator for each of the distributions, we chose not to use the original LUBM benchmark queries since they are *a*) geared towards reasoning systems and *b*) present a coarse grain of complexity in terms of composing triple patterns and number of unbound variables rendering them unsuitable for an in-depth evaluation of AVALANCHE. Instead we devised the 11 SPARQL queries of varying complexity listed in C (listings 5.5 through 6.10) based on the observation that the number of joins involved, their size (number of participating triple patterns), and type are important descriptors of a queries' potential complexity

and therefore induced effort. For example star joins can be executed in parallel as n-way joins reducing the complexity of such an operation. However, when joins are chained in a *read-after-write* manner one is forced to process them serially.

Consequently, queries $LQ_i, i \in [0, 10]$ are constructed in order of increased complexity by combining increasingly longer *read-after-write* join chains with increasingly larger sized star patterns.

Experiment #5: Varying Data Distribution The results of running all eleven queries on the three data distributions ($D1$, $D3$, and $D5$) are graphed in Figures 21 and 22. All runs are warm runs and each query was run 5 times. In addition to the default values set for all experiments the following AVALANCHE stopping configuration was used: *1*) a *stop sliding window* of size 3 plans, *2*) a number of 512 maximum concurrent asynchronous connections at any given moment, and *3*) a 0.01 bloom-filter false positive error rate.

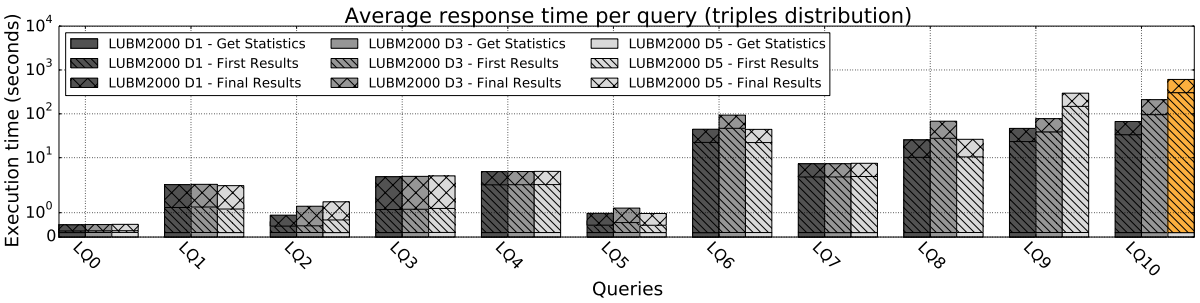


Fig. 21: Query execution times for all data distributions. Timeout cases are represented with orange.

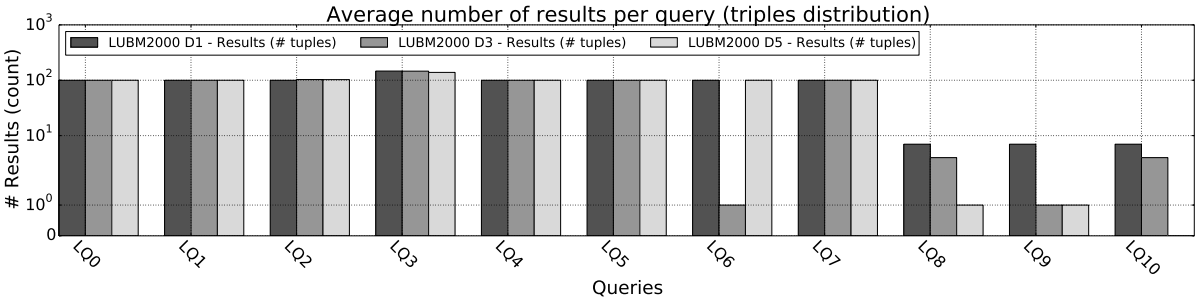


Fig. 22: Number of retrieved results (average) for all data distributions.

As can be observed in Figure 21 AVALANCHE exposes a relatively stable performance characteristic without timing-out for queries $LQ0$ through $LQ7$. Instance level spread is actually a benefiting factor for these queries that target replicated knowledge by providing more "chunks" of partial results, which in turn increases AVALANCHE's chances

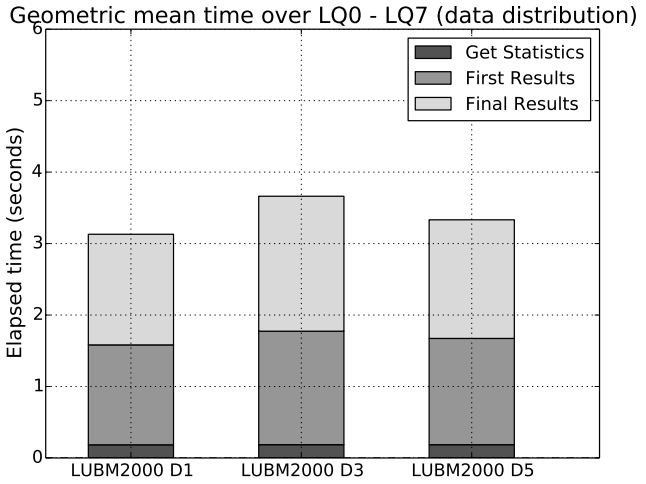


Fig. 23: Geometric mean of the execution time over all queries for $D1$,$D3$ and $D5$, queries $LQ0$ through $LQ7$.

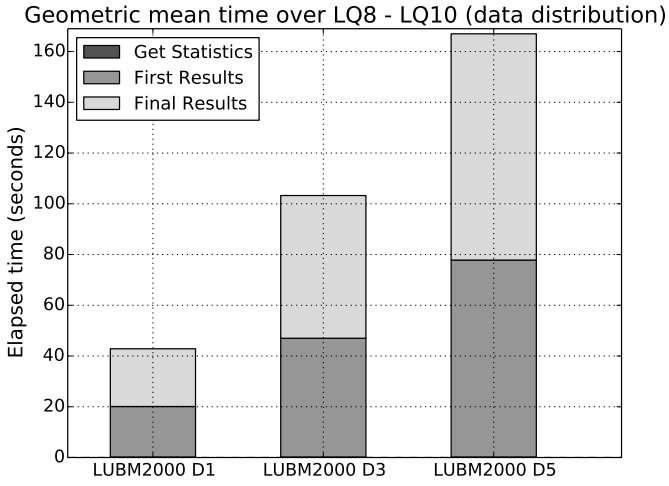


Fig. 24: Geometric mean of the execution time over all queries for $D1$,$D3$ and $D5$, queries $LQ8$ through $LQ10$.

of generating a "productive" plan. Looking at Figure 22, we can clearly observe that regardless of the degree of messiness (a university's triples spread to 1, 3 or 5 endpoints), AVALANCHE succeeds in retrieving about the same number of results exhibiting a highly stable behaviour. An exception is exhibited by $LQ6$ (Listing 6.6) where performance degrades only for distribution $D3$. This kind of system behaviour is expected in some cases, due mainly to the estimative nature of the cost model. In this particular case the first "productive" plan is discovered relatively late compared to the other 2 distribution cases.

Queries $LQ8$, $LQ9$ and $LQ10$ form a second group of queries. These queries target very specific knowledge pertinent to a single university leaving AVALANCHE with the task of identifying those endpoints (1, 3 or 5), which produce the desired result when combined. As can be observed, performance degrades dramatically with the number of hosts on which data is spread and with the number of joins generated by the query, i.e. query $FQ10$ times out (depicted in orange) for distribution $D5$ (triples spread over 5 endpoints). This result suggests that naïve selectivity estimation based cost models are not enough when dealing with *fine-grained triple-level messiness* at this scale, warranting novel and (more) accurate estimation statistics. Another effect of increased triples-spread is observed in the decline in recall for this second group of queries (Figure 22). A possible explanation for this observation is that as triples are distributed over more hosts, finding candidate joins becomes harder while the ones that are favoured first are usually the more selective and, hence, the ones with fewer results.

The systems' overall behaviour for the two query groups is observed more clearly in Figures 23 and 24, where the geometric mean over answering all queries against each distribution is shown. The Figures highlight the elapsed times for the three important execution phases in AVALANCHE. The *statistics gathering* phase accounts for a negligible part of the entire execution process and accounts to a mere 0.2 seconds on average for both query groups. We attribute this to the *Hexastore*-inspired read optimised indexing model of the RDF store used. We observe that AVALANCHE exposes a stable behaviour for the first group of queries finding first answers after an average of 1.5 seconds and completing the query after an average of 1.7 seconds. For the second group of queries, AVALANCHE exposes a slowdown effect in terms of finding first answers, retrieving them after an average of 48 seconds and completing the query after an average of 56 seconds. Finally, while AVALANCHE becomes slower it however, maintains its *robustness* as it will eventually find results.

5.3 Summary

Both evaluation settings in Sections 5.1 and 5.2 show AVALANCHE's stability against messiness. For the real world data-distribution setup based on Fedbench AVALANCHE was able to find first results in under one second for about 80% of the considered queries, while total results were retrieved under one second for about 70% of the queries, with the slowest running query taking about 5.5 seconds to complete. A notable exception is represented by query $FQ12$, which generates a large intermediate result set, potentially blocking or slowing down access to underlying shared resources like network connections

and database indexes. This is alleviated to some extent by: first, relying on asynchronous socket API's and second, isolating the execution of expensive queries/joins inside threads or processes. Other possibilities of reducing the overhead of expensive semi-joins is by compressing intermediate result sets. Even more, a good replacement strategy for semi-joins are bloom-joins, where the actual data sent is the bit-vector forming the bloom filter of the intermediate results set. The bloom-join is advantageous for large result sets as $sizeof(ResultSet_{subquery}) >> sizeof(BitVector_{bloomfilter})$.

Furthermore, as shown in the the third experiment when the broker-endpoints network latency changes then AVALANCHE's slowdown compared to the connection's slowdown exhibits a sub-linear characteristic as graphed in Figure 16. AVALANCHE is also able to dynamically adapt when some participating endpoints go offline when they are not the sole query results providers. Considering the synthetic LUBM dataset where a "brownian" spread of triples from their source host is simulated, AVALANCHE exhibits a high level of stability when answering queries that are selective with respect to knowledge that is likely to be replicated (i.e. classes) as seen in Figure 23. AVALANCHE does become progressively slower for queries that target specific resources (Figure 24). This happens since the objective functions considered do not leverage in any way the data distribution aspect.

6 Limitations, Optimizations, and Future Work

The work presented here exhibits two kinds of limitations. On one side the system could be extended and/or optimised; on the other side the external validity of the evaluation is limited. We will discuss both of these topics in turn.

System limitations and optimisations The AVALANCHE system has shown how a completely heterogeneous distributed query engine that makes no assumptions about data distribution could be implemented. The current approach does have a number of limitations as highlighted in Section 3, most notably the fact that it:

i) does not support UNION graph patterns,
ii) can be resource wasteful for some classes of query workloads, and
iii) does not offer result-set completeness guarantees.

UNIONS could be included by execution model as discussed in Section 3. One approach to address resource wastefulness would be to improve the quality of cost estimation, e.g., via learning. We intend to explore these avenues in future work. Result-set completeness is external to AVALANCHE and a characteristic of the Web-of-Data.

Furthermore, we need to better understand the cost-estimation functions used by the planner, investigate if the requirements put on participating triple-stores are reasonable, and empirically evaluate if the approach scales to an even larger number of active hosts.

To improve AVALANCHE's performance a number of research avenues and potential solutions stand out. For instance, the simplistic *source selection* algorithm can be improved with higher-quality statistics for a more accurate source selection process.

Another high-impact avenue is to enhance join estimation accuracy, i.e. by using *Bloom Filters*, histograms or schema-bound join predictive models which learn join distributions from previous observations. Moreover, we intend to investigate if a stateless approach is feasible since Avalanche currently assumes that remote endpoints keep partial results throughout plan execution to reduce local database operational cost. Note that the simple approach—the use of REST-ful services [Fielding and Taylor, 2002]—may not be applicable as the size of the state may be too large and overburden available bandwidth. Additionally, we will need to investigate how complex it would be in practice to generalise the notion of a *common key-space* beyond the textual representation of RDF terms in order to increase the performance of bandwidth-intensive join and merge operations.

Finally, we would like to point out that Avalanche completely ignores schema. Whilst this allows us to provide a schema-agnostic solution it does delegate the problem to the querying user. As a large number of publications on schema-integration [Euzenat and Shvaiko, 2007] and the owl:sameAs problem (i.e., [Halpin et al., 2010]) show a lot of work might still be needed to address this kind of messiness transparently. Hence, this is beyond the scope of Avalanche.

Evaluation limitations Our experiments rely on a limited number of physical resources available for accommodating the endpoints, the number of physical machines used is 4 to 16 times smaller than required in reality, where an endpoint would most often reside on an individual server. When one machine accommodates multiple endpoints, then these endpoints compete for shared resources (such as RAM, disk I/O, network I/O, and CPU-time). We think that the impact on our finding is mitigated by the choice of machines with more cores then endpoints. Furthermore, real-world endpoints would have to answers multiple query requests, each of which also competes for machine resources. Still, we believe that our setup is as realistic as possible in an experimental laboratory-setup and allows generalising the results.

In Avalanche we have so far focused on graph pattern matching and have thus ignored other SPARQL features like OPTIONAL and FILTER graph patterns. As part of our future work on Avalanche we intend to extend support to cover these features. Properly supporting FILTER graph patterns is likely to speed up query processing in Avalanche due to the intrinsic parallelism of union operations and due to the selective effect of filtering partial results – depending on how soon a FILTER can be evaluated.

7 Conclusion

In this paper we presented Avalanche, a novel approach for querying the Web-of-Data that (1) makes no assumptions about data distribution, availability, or partitioning exhibiting skew resistance for classes of queries that are selective with regards to replicated knowledge (i.e. Class information), (2) is dynamically adaptive to changing external network conditions, (3) provides up-to-date results, and (4) is flexible since it makes few limiting assumptions about the structure of participating triple stores. Specifically, we showed that Avalanche is able to execute non-trivial queries over distributed data-sources with an ex-ante unknown data-distribution. We showed that an extensible

cost model based on a common *Multi Objective Optimisation* method—the method of *Global Criterion*, where different heuristics can be plugged in without imposing changes to existing ones—can yield good performance in spite of different data distributions or changing latency while allowing for a messy Web-of-Data.

We designed AVALANCHE with the need to handle messy semi-structured data at large scales. The core idea follows the principle of *decentralisation*. It also supports *asynchrony* using asynchronous HTTP requests to avoid blocking, *autonomy* by delegating the coordination and execution of the distributed join/update/merge operations to the hosts, *concurrency* through the pipeline shown in Figure 3, *symmetry* by allowing each endpoint to act as the initiating AVALANCHE node for a query caller, as well as *fault tolerance* via proper exception and time-out handling and stopping conditions. By design AVALANCHE handles messiness generated by (i) schema alignment and data evolution, as AVALANCHE is schema agnostic its current view of the world is as a set of triples, (ii) data distribution through its extensible cost model, and (iii) source un-availability, as AVALANCHE dynamically dismisses plans issued to hosts that are not present anymore during the execution phase, still allowing other hosts (sources) to produce new and more results.

AVALANCHE's main limitation with respect to messiness is its assumption that participating data-sources are indexed (i.e., stored in some kind of triple store rather than "just" provided as files). In the light of its robustness against other kinds of messiness, however, we believe that AVALANCHE's capabilities outweigh this disadvantage—in particular since it would be simple to "wrap" any (known)file-based source with a combination of a triple-store and crawler.

To our knowledge, AVALANCHE is the first Semantic Web query system that makes no assumptions about the data distribution whatsoever. Whilst it is only a first implementation with a number of drawbacks it represents an important step towards querying a messy Web-of-Data by embracing its messiness as necessity (rather than an impediment) in order to foster its unpredictable growth.

Acknowledgements

This work was partially supported by the Swiss National Science Foundation under contract number 200021-118000. We are also grateful to the anonymous reviewers for their constructive comments, which helped to substantially improve the paper.

Appendix

A AVALANCHE Endpoint Operators

Execution Operators. For brevity, example query listings will not include the prefixes already defined in the motivating example query Q_{ex}.

`getTPCardinality(tp)`

As the name suggests, this operator is responsible with returning the number of instances matching the triple pattern *tp* on the callee endpoint. This operator is SPARQL (1.1) compliant and can be implemented in several fashions depending on wether the predicate is bound and VoID is used. To illustrate how, the following triple pattern from Q_{ex} is considered:
`< ?chebiDrug, chebi:image, ?chebiImage >.`

Example: getTPCardinality operator to SPARQL(1.1) mapping

```
PREFIX void: <http://rdfs.org/ns/void#>

## If predicate is bound and VoID is used
SELECT ?cardinality WHERE {
 ?dataset void:propertyPartition ?partition .
 ?partition void:property chebi:image .
 ?partition void:triples ?cardinality
}

## If VoID is not used but SPARQL 1.1 compliant
SELECT (COUNT(DISTINCT ?chebiDrug) as ?cardinality)  WHERE {
 ?chebiDrug chebi:image ?chebiImage
}
```

`getTotalTriples()`

SPARQL compliant as well, this is arguably the simplest operator. Its task being to report the total number of triples indexed by the endpoint. The overwhelming majority of modern day triple stores are aware of this fact and exposing this as a VoID statistic would be trivial.

Example: getTotalTriples operator to SPARQLmapping

```
## if VoID is used
SELECT ?dataset ?total  WHERE {
 ?dataset void:triples ?total .
}
```

`executeQuery(bgp, vars, values)`

This operator is virtually implemented by all RDF triple stores. The optional *vars* and *values* arguments are mapped directly to the `VALUES` term in SPARQL 1.1. For example consider the second fragment from Q_{ex} in Listing 6.2 with example dummy *values* for the `?drugBankName` variable:

Example: executeQuery operator to SPARQL1.1 mapping

```
SELECT ?chebiDrug ?chebiImage WHERE {
 ?chebiDrug chebi:image ?chebiImage .
 ?chebiDrug dc:title ?drugBankName
} VALUES (?drugBankName) {
 ("Drug A")
 ("Drug B")
 ("Drug C")
}
```

executeDistributedJoin(bgp_{local}, bgp_{remote}, *host*)
A critical part of the core functionality of any distributed database querying system is given by the ability to execute distributed joins. This operator is essentially a *proxy* operator as it relies on the ability to execute SPARQL queries both locally and remotely and functions as following: first the subquery bgp_{local} is executed locally as any regular SPARQL query. Next, the join variables (*vars*) between the two subqueries (bgp_{local} and bgp_{remote}) are determined and the partial results corresponding to them are selected (*values*). As the final step the executeQuery(bgp_{remote}, *vars*, *values*) operator is called on the remote *host*.

The following operator pair is optional and exists mainly for optimisation reasons. Their role is to simply reduce the end I/O cost of executing a distributed query:

executeDistributedReconciliation(bgp_{local}, bgp_{remote}, *host*)
Regarded as a "cleanup" operation the set-reconciliation procedure follows the execution of a distributed n-way join in order to remove partial results in excess resulting from preceding joins. Also a *proxy* operator baring a simplistic nature, its task is that of determining the *values* of the join *vars* between the two subqueries (bgp_{local} and bgp_{remote}) and calling executeReconciliation(bgp_{remote}, *vars*, *values*) on the remote *host*. Various optimizations are possible at this stage. Hence, instead of sending the actual set of *values* (compressed or not), a set of their hashes or a **bloom filter** can be employed, resulting in a hash- or a bloom filter-optimised distributed join.

executeReconciliation(*bgp*, *vars*, *values*)
Always called as the result of executing the executeDistributedReconciliation operator, its scope is to select and filter the excess results corresponding to the previously locally executed *bgp* query. As mentioned earlier this operator is designed to reduce the network traffic for the final *merge* phase of the distributed query execution. Depending on the optimisation mechanism used (hashing, bloom filters, or the actual set) the process can be exact or exhibit false positives (for bloom filters).

The following operators are required in the final stages of the query execution process:

executeDistributedMerge(bgp_{local}, bgp_{remote}, *host*)
Just like the previous executeDistributedJoin operator, this is also a proxy operator paired with executeMerge. The partial results contained in *results_table* corresponding to the previously executed query bgp_{local} are selected and sent remotely by calling executeMerge(bgp_{remote}, *results_table*) on *host*. This operator is outside the scope of SPARQL compliancy, however, it can be implemented as a simple HTTP GET call as described by the REST model [Fielding and Taylor, 2002].

executeMerge(*bgp*, *results_table*)
Called as a result of a distributed merge operation, this final operator in the execution pipeline implements the standard database INNER JOIN ($\bowtie$) operation on the incoming remote *results_table* and the local partial results table corresponding to the *bgp* query, which was previously executed during the distributed join phase.

materialise(*bgp*)
This operator is necessary when distributed joins are executed in a common ID space used by the remote endpoints to index RDF data-sets. As the name suggests its basic functionality is that of providing the mapping from ID to RDF literals, a necessary condition when formulating the final results.

State Management Operators. The following state management operators[26] are exposed by AVALANCHE as a means to allow query brokers to halt the distributed operations involved in answering a query when the desired results are found:

stopPlan(pid)
Although not strictly necessary for AVALANCHE to function, the operator ensures the "cleanup" and freeing of allocated resources while trying to satisfy a given plan denoted by the *pid* identifier (i.e. the MD5 hash of the SPARQL 1.1 query decomposition).

stopAllPlans(*Q*)
Similarly, the operator will stop the execution and free all resources allocated for the resolving of all plans pertaining to the considered query. To reduce network overhead the query string can be replaced with a simple hash of the actual query (i.e., the MD5 hash of the original SPARQL query).

[26] Both operators can be implemented as REST calls

B Fedbench Query Name Mapping

Table 5: Fedbench query name mapping

Collection	Fedbench Name	Name	Fedbench Name	Name
Cross Domain	CD 1a[c]	**FQ0**	CD 1b[c]	**FQ1**
	CD 2	**FQ2**	CD 3	**FQ3**
	CD 4	**FQ4**	CD 5	**FQ5**
	CD 6	**FQ6**	CD 7	**FQ7**
Life Sciences	LS 1a[c]	**FQ8**	LS 1b[c]	**FQ9**
	LS 2a[c]	**FQ10**	LS 2b[c]	**FQ11**
	LS 3	**FQ12**	LS 4	**FQ13**
	LS 5	**FQ14**	LS 6	**FQ15**
Life Sciences +[b]		**FQ33**		**FQ34**
		FQ35		**FQ36**
		FQ37		
Linked Data	LD 1	**FQ16**	LD 2	**FQ17**
	LD 3	**FQ18**	LD 4	**FQ19**
	LD 5	**FQ20**	LD 6	**FQ21**
	LD 7	**FQ22**	LD 8	**FQ23**
	LD 9	**FQ24**	LD 10	**FQ25**
	LD 11	**FQ26**		
SP^2Bench	SP2Bench Q1	**FQ27**	SP2Bench Q2[d]	**FQ28**
	SP2Bench Q5	**FQ29**	SP2Bench Q9a[c]	**FQ30**
	SP2Bench Q9b[c]	**FQ31**	SP2Bench Q10	**FQ32**

[a] Original query names from the Fedbench project: http://code.google.com/p/fbench/wiki/Queries.
[b] These queries are not part of the original Fedbench benchmark and therefore do not have a corresponding denomination. They are added for their increased complexity.
[c] Queries whose names are suffixed with *a* or *b* represent Fedbench queries that contain UNION graph patterns. The two subqueries are executed independently.
[d] Since the version of Avalanche used for this evaluation does not support the OPTIONAL graph pattern modifier, any OPTIONAL graph patterns were discarded from the query.

C LUBM Benchmark Queries

```
PREFIX lubm: <http://www.lehigh.edu/zhp2/2004/0401/univ-bench.owl#>PREFIX uni0:
    <http://www.Department1.University0.edu/>
```

Listing 5.4: PREFIXES

```
SELECT ?professor WHERE {
 ?professor lubm:name "FullProfessor1"} LIMIT 100
```

Listing 5.5: LQ0

```
SELECT ?department ?researchGroups WHERE {
 ?researchGroups lubm:subOrganizationOf ?department.
 ?department lubm:name "Department1"} LIMIT 100
```

Listing 5.6: LQ1

```
SELECT ?studentName WHERE {
 ?student lubm:name ?studentName.
 ?student lubm:memberOf <http://www.Department1.University0.edu>} LIMIT 100
```

Listing 5.7: LQ2

```
SELECT ?property ?value WHERE {
 ?professor lubm:name "FullProfessor1".
 ?professor ?property ?value} LIMIT 100
```

Listing 5.8: LQ3

```
SELECT ?mail ?phone WHERE {
 ?professor lubm:emailAddress ?mail.
 ?professor lubm:telephone ?phone.
 ?professor lubm:name "FullProfessor1"} LIMIT 100
```

Listing 5.9: LQ4

```
SELECT ?mail ?phone ?doctor WHERE {
 ?professor lubm:emailAddress ?mail.
 ?professor lubm:telephone ?phone.
 ?professor lubm:doctoralDegreeFrom ?doctor.
 ?professor lubm:name "FullProfessor1"} LIMIT 100
```

Listing 5.10: LQ5

```
SELECT ?studentName ?courseName WHERE {
 ?student lubm:takesCourse ?course.
 ?course lubm:name ?courseName.
 ?student lubm:name ?studentName.
 ?student lubm:memberOf <http://www.Department1.University0.edu>} LIMIT 100
```

Listing 5.11: LQ6

```
SELECT ?publication ?author ?department ?university WHERE {
 ?publication lubm:name "Publication0".
 ?publication lubm:publicationAuthor ?author.
 ?author lubm:worksFor ?department.
 ?department lubm:subOrganizationOf ?university} LIMIT 100
```

Listing 5.12: LQ7

```
SELECT ?name ?advisor ?department WHERE {
 ?advisor lubm:worksFor ?department.
 ?student lubm:advisor ?advisor.
 ?student lubm:name ?name.
 ?student lubm:takesCourse uni0:GraduateCourse33} LIMIT 100
```

Listing 5.13: LQ8

```
SELECT ?name ?tel ?advisor ?department WHERE {
 ?advisor lubm:worksFor ?department.
 ?student lubm:advisor ?advisor.
 ?student lubm:name ?name.
 ?student lubm:telephone ?tel.
 ?student lubm:takesCourse uni0:GraduateCourse33} LIMIT 100
```

Listing 5.14: LQ9

```
SELECT ?university ?student ?name ?tel WHERE {
 ?student lubm:advisor ?advisor.
 ?advisor lubm:worksFor ?department.
 ?department lubm:subOrganizationOf ?university.
 ?student lubm:name ?name.
 ?student lubm:telephone ?tel.
 ?student lubm:takesCourse uni0:GraduateCourse33} LIMIT 100
```

Listing 5.15: LQ10

D Fedbench Benchmark Queries

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX kegg: <http://bio2rdf.org/ns/kegg#>
PREFIX nytimes: <http://data.nytimes.com/elements/>
PREFIX geonames: <http://www.geonames.org/ontology#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX swc: <http://data.semanticweb.org/ns/swc/ontology#>
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX bench: <http://localhost/vocabulary/bench/>
PREFIX drugbank: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/>
PREFIX person: <http://localhost/persons/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbpedia: <http://dbpedia.org/resource/>
PREFIX swrc: <http://swrc.ontoware.org/ontology#>
PREFIX drugbank-category: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/
    drugcategory/>
PREFIX drugbank-drugs: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugs/>
PREFIX linkedmdb: <http://data.linkedmdb.org/resource/movie/>
PREFIX chebi: <http://bio2rdf.org/ns/bio2rdf#>
PREFIX purl: <http://purl.org/dc/terms/>
```

Listing 5.16: PREFIXES

```
SELECT ?predicate ?object WHERE {
 dbpedia:Barack_Obama ?predicate ?object}
```

Listing 5.17: FQ0

```
SELECT ?predicate ?object WHERE {
 ?subject owl:sameAs dbpedia:Barack_Obama.
 ?subject ?predicate ?object}
```

Listing 5.18: FQ1

```
SELECT ?party ?page WHERE {
 dbpedia:Barack_Obama dbpedia-owl:party ?party.
 ?x nytimes:topicPage ?page.
 ?x owl:sameAs dbpedia:Barack_Obama}
```

Listing 5.19: FQ2

```
SELECT ?president ?party ?x WHERE {
 ?president rdf:type dbpedia-owl:President.
 ?president dbpedia-owl:nationality dbpedia:United_States.
 ?president dbpedia-owl:party ?party.
 ?x nytimes:topicPage ?page.
 ?x owl:sameAs ?president}
```

Listing 5.20: FQ3

```
SELECT ?actor ?news WHERE {
 ?film purl:title "Tarzan".
 ?film linkedmdb:actor ?actor.
 ?actor owl:sameAs ?x.
 ?y owl:sameAs ?x.
 ?y nytimes:topicPage ?news}
```

Listing 5.21: FQ4

```
SELECT ?film ?director ?genre WHERE {
 ?film dbpedia-owl:director ?director.
 ?director dbpedia-owl:nationality dbpedia:Italy.
 ?x owl:sameAs ?film.
 ?x linkedmdb:genre ?genre}
```

Listing 5.22: FQ5

```
SELECT ?name ?location WHERE {
 ?artist foaf:name ?name.
 ?artist foaf:based_near ?location.
 ?location geonames:parentFeature ?germany.
 ?germany geonames:name "Federal Republic of Germany"}
```

Listing 5.23: FQ6

```
SELECT ?location ?news WHERE {
 ?location geonames:parentFeature ?parent.
 ?parent geonames:name "California".
 ?y owl:sameAs ?location.
 ?y nytimes:topicPage ?news}
```

Listing 5.24: FQ7

```
SELECT ?drug ?melt WHERE {
 ?drug drugbank:meltingPoint ?melt}
```

Listing 5.25: FQ8

```
SELECT ?drug ?melt WHERE {
 ?drug dbpedia-owl:drug/meltingPoint ?melt}
```

Listing 5.26: FQ9

```
SELECT ?predicate ?object WHERE {
 drugbank-drugs:DB00201 ?predicate ?object}
```

Listing 5.27: FQ10

```
SELECT ?predicate ?object WHERE {
 drugbank-drugs:DB00201 owl:sameAs ?caff.
 ?caff ?predicate ?object}
```

Listing 5.28: FQ11

```
SELECT ?Drug ?IntDrug ?IntEffect WHERE {
 ?Drug rdf:type dbpedia-owl:Drug.
 ?y owl:sameAs ?Drug.
 ?Int drugbank:interactionDrug1 ?y.
 ?Int drugbank:interactionDrug2 ?IntDrug.
 ?Int drugbank:text ?IntEffect}
```

Listing 5.29: FQ12

```
SELECT ?drugDesc ?cpd ?equation WHERE {
 ?drug drugbank:drugCategory drugbank-category:cathartics.
 ?drug drugbank:keggCompoundId ?cpd.
 ?drug drugbank:description ?drugDesc.
 ?enzyme kegg:xSubstrate ?cpd.
 ?enzyme rdf:type kegg:Enzyme.
 ?reaction kegg:xEnzyme ?enzyme.
 ?reaction kegg:equation ?equation}
```

Listing 5.30: FQ13

```
SELECT ?drug ?keggUrl ?chebiImage WHERE {
 ?drug rdf:type drugbank:drugs.
 ?drug drugbank:keggCompoundId ?keggDrug.
 ?keggDrug chebi:url ?keggUrl.
 ?drug drugbank:genericName ?drugBankName.
 ?chebiDrug dc:title ?drugBankName.
 ?chebiDrug chebi:image ?chebiImage}
```

Listing 5.31: FQ14

```
SELECT ?drug ?title WHERE {
 ?drug drugbank:drugCategory drugbank-category:micronutrient.
 ?drug drugbank:casRegistryNumber ?id.
 ?keggDrug rdf:type kegg:Drug.
 ?keggDrug chebi:xRef ?id.
 ?keggDrug dc:title ?title}
```

Listing 5.32: FQ15

```
SELECT ?paper ?p ?n WHERE {
 ?paper swc:isPartOf <http://data.semanticweb.org/conference/iswc/2008/
    poster_demo_proceedings>.
 ?paper swrc:author ?p.
 ?p rdfs:label ?n}
```

Listing 5.33: FQ16

```
SELECT ?proceedings ?paper ?p WHERE {
 ?proceedings swc:relatedToEvent <http://data.semanticweb.org/conference/eswc/2010
    >.
 ?paper swc:isPartOf ?proceedings.
 ?paper swrc:author ?p}
```

Listing 5.34: FQ17

```
SELECT ?paper ?p ?x ?n WHERE {
 ?paper swc:isPartOf <http://data.semanticweb.org/conference/iswc/2008/
    poster_demo_proceedings>.
 ?paper swrc:author ?p.
 ?p owl:sameAs ?x.
 ?p rdfs:label ?n}
```

Listing 5.35: FQ18

```
SELECT ?role ?p ?paper ?proceedings WHERE {
 ?role swc:isRoleAt <http://data.semanticweb.org/conference/eswc/2010>.
 ?role swc:heldBy ?p.
 ?paper swrc:author ?p.
 ?paper swc:isPartOf ?proceedings.
 ?proceedings swc:relatedToEvent <http://data.semanticweb.org/conference/eswc/2010
    >}
```

Listing 5.36: FQ19

```
SELECT ?a ?n WHERE {
 ?a dbpedia-owl:artist dbpedia:Michael_Jackson.
 ?a rdf:type dbpedia-owl:Album.
 ?a foaf:name ?n}
```

Listing 5.37: FQ20

```
SELECT ?director ?film ?x ?y ?n WHERE {
 ?director dbpedia-owl:nationality dbpedia:Italy.
 ?film dbpedia-owl:director ?director.
 ?x owl:sameAs ?film.
 ?x foaf:based_near ?y.
 ?y geonames:officialName ?n}
```

Listing 5.38: FQ21

```
SELECT ?x ?n WHERE {
 ?x geonames:parentFeature <http://sws.geonames.org/2921044/>.
 ?x geonames:name ?n}
```

Listing 5.39: FQ22

```
SELECT ?drug ?id ?s ?o ?sub WHERE {
 ?drug drugbank:drugCategory drugbank-category:micronutrient.
 ?drug drugbank:casRegistryNumber ?id.
 ?drug owl:sameAs ?s.
 ?s foaf:name ?o.
 ?s skos:subject ?sub}
```

Listing 5.40: FQ23

```
SELECT ?x ?p WHERE {
 ?x skos:subject dbpedia:Category:FIFA_World_Cup-winning_countries.
 ?p dbpedia-owl:managerClub ?x.
```

```
 ?p foaf:name "Luiz Felipe Scolari"}
```

Listing 5.41: FQ24

```
SELECT ?n ?p2 ?u WHERE {
 ?n skos:subject dbpedia:Category:Chancellors_of_Germany.
 ?n owl:sameAs ?p2.
 ?p2 nytimes:latest_use ?u}
```

Listing 5.42: FQ25

```
SELECT ?x ?y ?d ?p ?l WHERE {
 ?x dbpedia-owl:team dbpedia:Eintracht_Frankfurt.
 ?x rdfs:label ?y.
 ?x dbpedia-owl:birthDate ?d.
 ?x dbpedia-owl:birthPlace ?p.
 ?p rdfs:label ?l}
```

Listing 5.43: FQ26

```
SELECT ?yr WHERE {
 ?journal rdf:type bench:Journal.
 ?journal dc:title "Journal 1 (1940)".
 ?journal purl:issued ?yr}
```

Listing 5.44: FQ27

```
SELECT ?inproc ?author ?booktitle ?title ?proc ?ee ?page ?url ?yr WHERE {
 ?inproc rdf:type bench:Inproceedings.
 ?inproc dc:creator ?author.
 ?inproc bench:booktitle ?booktitle.
 ?inproc dc:title ?title.
 ?inproc purl:partOf ?proc.
 ?inproc rdfs:seeAlso ?ee.
 ?inproc swrc:pages ?page.
 ?inproc foaf:homepage ?url.
 ?inproc purl:issued ?yr}
```

Listing 5.45: FQ28

```
SELECT ?person ?name WHERE {
 ?article rdf:type bench:Article.
 ?article dc:creator ?person.
 ?inproc rdf:type bench:Inproceedings.
 ?inproc dc:creator ?person.
 ?person foaf:name ?name}
```

Listing 5.46: FQ29

```
SELECT ?predicate WHERE {
 ?person rdf:type foaf:Person.
 ?subject ?predicate ?person}
```

Listing 5.47: FQ30

```
SELECT ?predicate WHERE {
 ?person rdf:type foaf:Person.
 ?person ?predicate ?object}
```

Listing 5.48: FQ31

```
SELECT ?subject ?predicate WHERE {
 ?subject ?predicate person:Paul_Erdoes}
```

Listing 5.49: FQ32

```
SELECT ?drug ?enzyme ?reaction WHERE {
 ?drug1 drugbank:drugCategory drugbank-category:antibiotics.
 ?drug2 drugbank:drugCategory drugbank-category:antiviralAgents.
 ?drug3 drugbank:drugCategory drugbank-category:antihypertensiveAgents.
 ?I1 drugbank:interactionDrug2 ?drug1.
 ?I1 drugbank:interactionDrug1 ?drug.
 ?I2 drugbank:interactionDrug2 ?drug2.
 ?I2 drugbank:interactionDrug1 ?drug.
 ?I3 drugbank:interactionDrug2 ?drug3.
 ?I3 drugbank:interactionDrug1 ?drug.
 ?drug owl:sameAs ?drug5.
 ?drug5 rdf:type dbpedia-owl:Drug.
 ?drug drugbank:keggCompoundId ?cpd.
 ?enzyme kegg:xSubstrate ?cpd.
 ?enzyme rdf:type kegg:Enzyme.
 ?reaction kegg:xEnzyme ?enzyme.
 ?reaction kegg:equation ?equation}
```

Listing 5.50: FQ33

```
SELECT ?drug ?drug1 ?drug2 ?drug3 ?drug4 WHERE {
 ?drug1 drugbank:drugCategory drugbank-category:antibiotics.
 ?drug2 drugbank:drugCategory drugbank-category:antiviralAgents.
 ?drug3 drugbank:drugCategory drugbank-category:antihypertensiveAgents.
 ?drug4 drugbank:drugCategory drugbank-category:anti-bacterialAgents.
 ?I1 drugbank:interactionDrug2 ?drug1.
 ?I1 drugbank:interactionDrug1 ?drug.
 ?I2 drugbank:interactionDrug2 ?drug2.
 ?I2 drugbank:interactionDrug1 ?drug.
 ?I3 drugbank:interactionDrug2 ?drug3.
 ?I3 drugbank:interactionDrug1 ?drug.
 ?I4 drugbank:interactionDrug2 ?drug4.
 ?I4 drugbank:interactionDrug1 ?drug}
```

Listing 5.51: FQ34

```
SELECT ?drug WHERE {
 ?drug1 drugbank:possibleDiseaseTarget <http://www4.wiwiss.fu-berlin.de/diseasome/
    resource/diseases/302>.
 ?drug2 drugbank:possibleDiseaseTarget <http://www4.wiwiss.fu-berlin.de/diseasome/
    resource/diseases/53>.
 ?drug3 drugbank:possibleDiseaseTarget <http://www4.wiwiss.fu-berlin.de/diseasome/
    resource/diseases/59>.
```

```
 ?drug4 drugbank:possibleDiseaseTarget <http://www4.wiwiss.fu-berlin.de/diseasome/
     resource/diseases/105>.
 ?I1 drugbank:interactionDrug2 ?drug1.
 ?I1 drugbank:interactionDrug1 ?drug.
 ?I2 drugbank:interactionDrug2 ?drug2.
 ?I2 drugbank:interactionDrug1 ?drug.
 ?I3 drugbank:interactionDrug2 ?drug3.
 ?I3 drugbank:interactionDrug1 ?drug.
 ?I4 drugbank:interactionDrug2 ?drug4.
 ?I4 drugbank:interactionDrug1 ?drug.
 ?drug drugbank:casRegistryNumber ?id.
 ?keggDrug rdf:type kegg:Drug.
 ?keggDrug chebi:xRef ?id.
 ?keggDrug dc:title ?title}
```

Listing 5.52: FQ35

```
SELECT ?d ?drug5 ?cpd ?enzyme ?equation WHERE {
 ?drug1 drugbank:possibleDiseaseTarget <http://www4.wiwiss.fu-berlin.de/diseasome/
     resource/diseases/261>.
 ?I1 drugbank:interactionDrug2 ?drug1.
 ?I1 drugbank:interactionDrug1 ?drug.
 ?drug drugbank:possibleDiseaseTarget ?d.
 ?drug owl:sameAs ?drug5.
 ?drug5 rdf:type dbpedia-owl:Drug.
 ?drug drugbank:keggCompoundId ?cpd.
 ?enzyme kegg:xSubstrate ?cpd.
 ?enzyme rdf:type kegg:Enzyme.
 ?reaction kegg:xEnzyme ?enzyme.
 ?reaction kegg:equation ?equation}
```

Listing 5.53: FQ36

```
SELECT ?drug5 ?drug6 WHERE {
 ?drug1 drugbank:possibleDiseaseTarget <http://www4.wiwiss.fu-berlin.de/diseasome/
     resource/diseases/319>.
 ?drug1 drugbank:possibleDiseaseTarget <http://www4.wiwiss.fu-berlin.de/diseasome/
     resource/diseases/270>.
 ?I1 drugbank:interactionDrug1 ?drug1.
 ?I1 drugbank:interactionDrug2 ?drug.
 ?drug1 owl:sameAs ?drug5.
 ?drug owl:sameAs ?drug6}
```

Listing 5.54: FQ37

Chapter 6

Design Enhancements & Optimisations at Scale

This chapter is based on a submission to the:

Journal of Semantic Web – Interoperability, Usability, Applicability
under submission at the time of writing

x-Avalanche: Optimisation Techniques for Large Scale Federated SPARQL Query Processing

Cosmin A. Başca and Abraham Bernstein

DDIS, Department of Informatics, University of Zurich, Zurich, Switzerland
basca@ifi.uzh.ch, bernstein@ifi.uzh.ch

Abstract. Attributes like ease of linking and integration, flexibility and standardisation are making the RDF data model more popular. As a consequence, more RDF data gets published across different domains. This distributed publication of RDF data ethos embodies the spirit of the Web of Data. While centralised RDF storage has gotten more scalable in order to keep up with the increase of published data, the problem of querying large federations of RDF datasets has not received as much attention.
In this paper we extend our existing AVALANCHE federation engine to address some of the most pertinent issues with federated RDF query processing. First, we add *support for disjunctions* by employing a distributed union operator capable of scaling to hundreds or thousands of endpoints. Second, we *enhance the distributed state management with remote caches* aimed to reduce the high latency typical of SPARQL endpoints. Finally, we introduce a *novel and parallel-friendly optimisation paradigm* designed not only to offer an optimal tradeoff between total query execution time and fast first results, but to also consider an extended planning space unexplored so far.
Our results show that combined, these capabilities improve our system's performance by up to 70 times over the best performing SPARQL federation engine and find an optimal performance tradeoff between delivering first results and total query execution time under external constraints.

1 Introduction

In recent years, the RDF data model has received more attention; primarily due to factors that revolve around the data models' flexibility and standardisation. Linking RDF datasets as well as extending them wether with new data, annotations, or new versions is easy. Additionally, the semi-structured format is a natural fit for storing and representing graph data. As a consequence , the amount of published RDF continues to grow steadily. To cope with the growth of individual datasets—for example computational biology RDF datasets can amass to billions of triples such as *uniprot.org*, which has 6.95 billion triples—centralised indexing and storage solutions are becoming more scalable. At the same time the number of RDF datasets also continues to grow, as partly shown by the evolution of the *Linked Open Data* (LoD) cloud[1]. However, unlike centralised storage systems, federated RDF engines have not seen much attention while often providing limited support for the SPARQL 1.1 federation extensions.

1.1 Motivation

Over the years, substantial research has been carried out to address performance issues, location transparency, and to improve the SPARQL 1.1 federation specification Acosta

[1] http://lod-cloud.net/

et al. [2011], Başca and Bernstein [2010, 2014], Görlitz and Staab [2011], Quilitz and Leser [2008], Saleem et al. [2014], Schwarte et al. [2011]. These systems have primarily focused on addressing performance issues that are endemic to the LoD ecosystem. However, not all problematic aspects have been addressed to the same extent. One such issue stems from the LoD's schema richness and broad semantic diversity. In this setup, typical real-world and benchmark queries like the ones from FedBench Schmidt et al. [2011] are *semantically selective* — i.e., the vocabularies bound to the query restrict the execution of the query to only a few endpoints, considerably reducing the size of the problem. Having to deal with only a handful of endpoints at a time simplifies the position of typical SPARQL federation engines. The limitation to of investigations to semantically selective situations can lead to a lack of attention and optimisations that target more difficult scenarios.

There are cases in which the implicit assumption of *semantic selectivity* does not hold. First, it is foreseeable that as the size of the published RDF data continues to grow so is the number of endpoints that are *semantically homogenous*, i.e., store data with the same schema. Second, given the "messiness" of the LoD, which stems from the use of similar yet overlapping vocabularies, it is not uncommon to rewrite SPARQL queries in order to capture more of the potentially relevant data. For these scenarios, the large size of the problem requires:

a) novel and scalable system designs that are not addressed by current methods and standards,
b) novel query optimisation strategies, and
c) updated and comprehensive benchmarks designed to capture the issues of large RDF federations.

Equally important, flexibility has to be taken into account. A flexibly designed RDF federation engine must be compatible to a large degree with the existing SPARQL 1.1 standard and make few or no assumptions about the underlying RDF storage technology.

1.2 Contributions

In this paper we present novel methods, architectural enhancements, and optimisations for federated RDF engines, which, when combined, offer dramatic performance improvements over existing approaches while at the same time maintaining a flexible design.

To ascertain the validity of our hypotheses we fully implemented the methods by extending the Avalanche SPARQL federation engine Başca and Bernstein [2014]. We refer to the extension as x-Avalanche. Specifically, the technical contributions of this paper can be grouped into *Query Execution & Operator Design*, *Optimisation*, and *Implementation & Evaluation*. They are as follows:

Optimisation

1. We propose a novel approach to optimally reduce and explore an *extended planning space* for large federations of SPARQL endpoints (that has a parametric and non-parametric variant), where data is partitioned. We also show how to optimally find

the largest partial result-set that can be retrieved in the shortest possible time under user / domain defined constraints and given the cost model.
2. We identify a new class of easily parallelizable plans we call *fragmented bushy plans* – the top level logical node is a disjunction of standard plan subtrees.

Query Execution & Operator Design

3. We introduce a novel parallel union operator scalable to hundreds or thousands of endpoints.
4. We present an extended distributed state management protocol with support for disjunctions. Each operator is designed to execute directly or by proxy, i.e., delegate the operator's execution to a remote endpoint. All query execution X-AVALANCHE operators rely only on the SPARQL 1.1 protocol.
5. We show that a distributed caching strategy tailored for federated SPARQL queries is able to mitigate to a significant extent the high latency typical of SPARQL endpoints.

Implementation & Evaluation

6. We propose a simple synthetic benchmark based on LUBM Guo et al. [2005] and the design of the Waterloo SPARQL Diversity Test Suite or WatDiv Aluç et al. [2014] with support for different data distributions. We provide an open source implementation of that benchmark in the *rdftools*[2] project, also containing a description of the queries[3].
7. We present the implementation of the X-AVALANCHE system and *performance measurements* against FedX a state of the art top performing federated SPARQL engine Saleem et al. [2014], with support for *location transparency*.

The remainder of this paper is structured as follows. Section 2 describes state of the art federated SPARQL query approaches and optimality guarantee optimisation methods. A scalable union operator is introduced in Section 4, while the design decisions for X-AVALANCHE's extended query execution protocol are discussed in Section 5. A detailed evaluation of X-AVALANCHE follows in Section 6. We discuss limitations and future work directions in Section 7 and conclude in Section 8.

2 Background

In the following section we describe related and similar works to our system X-AVALANCHE. They can be grouped into: *federated SPARQL processing* and *query optimisation*. We also briefly describe the original AVALANCHE federation engine.

2.1 Related Work

Federated SPARQL Processing The continuous growth of the Web of Data (WoD) has given rise to new opportunities and challenges in querying this global repository of

[2] https://github.com/cosminbasca/rdftools
[3] https://github.com/cosminbasca/rdftools/blob/master/doc/DESCRIPTION.md

distributed but interlinked datasets. The Linked Open Data (LoD)[4] alone amassed over 60 billion assertions spread over more than 1000 datasets spanning a broad spectrum of domains. Typically, data on the LoD is shared either by following W3C's Linked Data[5] guidelines, indexed and exposed via a SPARQL endpoint, or simply available as compressed data dumps. While querying the WoD has seen much attention, in the following we will focus only on federations of SPARQL endpoints, such as those querying the indexed LoD, but not limited to.

One of the earliest approaches to offer *location transparency* materialised in DARQ Quilitz and Leser [2008]. Since the SPARQL 1.1 federation extensions were standardised much later, the authors relied on their own RDF-based representation of *service descriptions*. These provided a declarative way to describe the indexed data alongside useful statistics, which were valuable during query optimisation. A second wave of research has given birth to several more SPARQL federation engines. In FedX Schwarte et al. [2011], another virtual integrator of SPARQL endpoints, the authors develop new join execution strategies designed to minimise the number of requests sent to participating endpoints. Unlike FedX which makes use of a *rule-based* or *heuristic* query optimiser, SPLENDID Görlitz and Staab [2011] features a Dynamic Programming (DP) cost based optimiser able to guarantee plan optimality – within the confines of the cost model. The authors overcome one of the major impediments to using traditional database techniques for federated SPARQL processing by extracting advanced statistics from voID[6] endpoint descriptors. When voID statistics are not available, SPLENDID reverts to using ASK queries when selecting source endpoints.

A series of factors endemic to the WoD such as *i*) uncontrollable network conditions, i.e., no guarantees can be made about latency, bandwidth or availability, *ii*) inaccurate statistics, i.e., continuous data growth in both number of datasets and size, as well as *iii*) dynamic data and workload, have prompted the adoption to various degrees of *adaptive query processing* methods. For example, ADERIS Lynden et al. [2011] a mediator based federation, utilises adaptive join reordering given a predefined cost model. ANAPSID Acosta et al. [2011] is adaptive during query execution as well as during source selection. Exhibiting an intra-operator flavour of adaptivity the system features a non-blocking operator design. In contrast, Avalanche Başca and Bernstein [2010, 2014] features an inter-operator adaptive query execution design. Statistics about cardinalities and data distribution are obtained before query execution and used during optimisation. The system uses a fragmented execution model were top-k partial plans are executed concurrently, until user defined termination conditions are met or the plan fragment space is exhausted.

Given the sheer size of the LoD, recent research into federated SPARQL querying has focused more on the parallelism aspect of query processing. For instance, LHD Wang et al. [2013] like previous systems makes use of a variant of the popular selectivity based cost model, coupled with a parallel execution system that exploits streaming in order to

[4] http://stats.lod2.eu/
[5] http://www.w3.org/TR/ldp-bp/
[6] http://www.w3.org/TR/void/

minimise query execution time. An extension of FedX, FedSearch Nikolov et al. [2013] a hybrid federation search engine, is designed to execute combined structured SPARQL queries with full-text search. The system employs on-the-fly adaptation of the query plan and is optimised to execute top-k hybrid search queries over multiple data sources. Finally, in Saleem et al. [2013] the authors focus on the problem of duplicate data on the WoD. The proposed method, DAW which is used to extend the DARQ, SPLENDID and FedX federation engines, shows great promise in reducing the number of queries sent to endpoints.

Query Optimisation The ideal query optimiser would feature the lowest optimisation time (a small search space) and optimal plans. In the centralised case, the size of the search space is primarily governed by the number of joins in the query. Of secondary concern, the shape of the query can be used to further reduce the search space, i.e., leverage the fact that the query contains star-patterns.[7] When resolving complex federated SPARQL queries, query optimisers typically switch to a rule-based mode of operation in order to cope with the large planning space and answer the query in reasonable time. This is undesirable since *1*) the optimiser is forced to *drop any optimality guarantees* and *2*) using heuristics worsens the problem of *accurately estimating the cost* of complex queries Ioannidis and Christodoulakis [1991].

As analysed in previous works Ono and Lohman [1990], the time complexity of a DP optimiser in a centralised DBMS is $\mathcal{O}(3^n)$, where n is the number *triple patterns*. Similarly, the space complexity is $\mathcal{O}(2^n)$. One way to reduce the optimisation time is to adapt the general DP approach as described in Kossmann and Stocker [2000] by applying DP several times iteratively, while optimising the query. The method is known as Iterative Dynamic Programming (IDP) and features a reasonable polynomial time complexity, but does not guarantee overall optimal plans when more than one iteration of DP is performed. It does, however, find the optimal plan under the imposed resource constrains. The number of iterations can be controlled by the database administrator or adjusted automatically considering resource allocation (e.g., memory or time). In a distributed context, when data is replicated at different sites, the size of the search space explodes in the worst case. In this case the time complexity of a classic DP optimiser is $\mathcal{O}(s^3 * 3^n)$, while its space complexity falls into the $\mathcal{O}(s * 2^n + s^3)$ class, where s represents the number of sites that hold data.

For partitioned setups, however, traditional DP optimisers consider partitions usually at the leaf nodes as physical unions between sites, conveying the advantage of a reduced planning space over the replicated setup. In doing so, however, an *extended planning space* that can contain better join and union orderings is left unexplored, as this would again lead to an explosion of the search space. A first attempt to partially explore this *extended planning space* is presented in Herodotou et al. [2011], where the authors propose a method suitable for both centralised and parallel relational DBMS'. The central innovation of the algorithm is the introduction of a clustering phase aimed at discarding unnecessary child table (partitions) joins during planning, when information

[7] star-patterns are common when retrieving resource attributes

about which partitions can join is present. While this method is applicable for *range, list* or *hash* partitioning schemes typically encountered in such systems, it is not suitable for the global and uncontrollable nature of the Web of Data.

2.2 Avalanche

Here, we give a brief overview of our previous Avalanche federated SPARQL engine. Avalanche's architecture is organised to accommodate a three phase execution model. First, relevant endpoints are identified. Second, query specific cardinalities are retrieved and finally the query planning and execution phases follow. While finding out the cardinality of a basic graph pattern (BGP) can be expensive operation for an RDF store, aggressive indexing techniques like the ones implemented by RDF-3X Neumann and Weikum [2010] and Hexastore Weiss et al. [2008] or high performance implementations such as Virtuoso[8] allow for fast retrieval of *triple pattern cardinalities*. Furthermore, voID[9] Alexander et al. [2009] descriptions of the indexed data can accomplish the same. If no catastrophic SPARQL endpoint failures occur, the system is *eventually complete*.

Tailored to address the semantic heterogeneity and lack of guarantees that are characteristic to the WoD, Avalanche employs a *fragmented* execution model. Here, the query is decomposed into the union of all *query fragments*. A query fragment, or fragment in short, is defined as the conjunction of all query triple patterns with the restriction that a triple pattern can be resolved by one host, i.e., no disjunctions allowed inside a fragment. Avalanche enumerates all fragments using a priority queue based repeated depth first search traversal algorithm. This allows Avalanche to *1*) generate fragments in a given order, i.e., favour faster and productive fragments and *2*) execute all or K fragments concurrently at any given moment, i.e., dynamically adapting to network conditions and endpoint availability.

3 Optimisation

Currently, the LoD exhibits high semantic selectivity and limited dataset partitioning. This is primarily due to its schema richness and broad semantic diversity, which leads to a drastic reduction of the number of participating SPARQL endpoints when querying. Designed to target the current state of the LoD, the original Avalanche federation engine exhibits a number of shortcomings. First, while it features a multi-fragment concurrent execution model, it does not provide support for disjunctions. Second, Avalanche does not statically optimise each plan fragment and instead it employs a non-optimal *greedy execution strategy* (GRDY), where the order of each join is decided on the fly. Finally, Avalanche makes use of a selectivity based cost model to decide on the order in which fragments are generated, and does not attempt to reduce the size of the planning space.

However, as the size of the LoD continues to grow, we expect to see a decrease in semantic selectivity and an increase of dataset partitioning. Consequently, new solutions

[8] https://github.com/openlink/virtuoso-opensource
[9] http://www.w3.org/TR/void/

are warranted and therefore, this section introduces new optimisation approaches that are more suitable for these scenarios, optimisations which are built into X-AVALANCHE – an extension of AVALANCHE.

3.1 Cost Model and Optimisation Strategies

X-AVALANCHE is designed to address large RDF federations where data is horizontally partitioned between many endpoints and where *semantic selectivity* has a negligible or low impact. In other words it is designed to deal with large and semantically homogenous distributed datasets. In order to proceed further, we relax the notion of a *plan fragment* as used by AVALANCHE and redefine it as follows:

Definition 1. *A **plan fragment** is a query plan for which only a subset of all participating sites* **s** *are considered.*

In other words, a (conjunctive) plan fragment can also contain disjunctions or unions. Defined as such (Definition 1), a query can have between 1 ($\forall$ triple pattern bound to all sites) and s^n fragments ($\forall$ triple pattern bound to one site).

A first enhancement over AVALANCHE is that each fragment can now be statically optimised in contrast to the greedy execution strategy. For this purpose we employ the classic *dynamic programming* (DP) Bellman [1957] method. Since DP features a worst case time complexity of $\mathcal{O}(3^n)$ Ono and Lohman [1990] for n triple patterns, we consider the following simplifying assumptions in order to reduce the plan space:

- Like in System R Astrahan et al. [1976] we explore only *left-deep* plan trees and avoid cross-products whenever possible.
- The order of the join operands is ignored during the planning phase and determined at runtime, i.e., always ship the smallest bindings set.

Furthermore, since network communication introduces the highest latency during query execution, we rely on the simplifying assumption that *the number of partial results has the highest impact on performance*. Therefore, we base the cost model used to optimise each plan fragment on the estimation of the query's selectivity Stocker et al. [2008b]. Equations 1 and 2 show how the cardinality of joining and unioning two triple patterns tp_1 and tp_2 is estimated, where Θ represents the total number of triples.

$$|tp_1 \bowtie tp_2| = |tp_1| \times |tp_2| \times \frac{|tp_1| + |tp_2|}{2 \times \Theta}, \tag{1}$$

$$|tp_1 \cup tp_2| = \max(|tp_1|, |tp_2|) \tag{2}$$

While naïve, this model of partial result size estimation has the advantage that no other statistics are needed aside from triple pattern cardinalities or estimates of. However, this comes at a cost: less accurate estimations can, in practice, render some plans much more expensive than estimated. As results from Section 6 show, this model, while simple, was able to dramatically improve performance over a top performing state of the art approach.

3.2 Extended Space Reduction

To show how the optimal partition-aware union grouping method works, we reuse the notion of *plan matrix* from Başca and Bernstein [2014]. The *plan matrix* or $\mathcal{PM}$ is a compact representation of the cardinalities of all query triple patterns on all sites as follows:

$$\mathcal{PM} = \begin{bmatrix} card_{0,0} & \cdots & card_{0,n} \\ \vdots & \ddots & \vdots \\ card_{s,0} & \cdots & card_{s,n} \end{bmatrix} \tag{3}$$

where s and n represent the number of sites and triple patterns respectively, while $card_{i,j}$ is the cardinality[10] of triple pattern i on site j. For simplicity, the remainder of this paper only considers plans that are constructed with conjunctions ($\bowtie$) and disjunctions ($\cup$). While not a trivial matter and outside the scope of this work, support for OPTIONAL and FILTER graph patterns could be provided by relying for example on their mapping to relational algebra operators Cyganiak [2005].

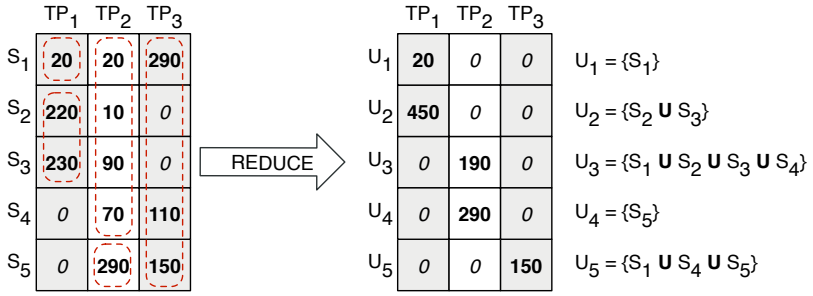


Fig. 1: Example $\mathcal{PM}$ and a possible reduced $\mathcal{PM}^*$. S_i represent the sites holding data, while TP_j represent *triple-patterns*.

Note that a plan matrix $\mathcal{PM}$ of size (s, n) may lead to up to s^n plan fragments. Consider for a moment the unlikely case, where each triple pattern tp_i can be matched on every site s_j (i.e., $\mathcal{PM}$ contains no zeros). A valid plan fragment can now be constructed by choosing one of the sites for each triple pattern. As there are s sites to choose from, there are s valid choices for each of the n triple patterns resulting in s^n possible fragments. Obviously, this plan space is too large in the worst case. Heuristics-based algorithms circumvent the problem of a large $\mathcal{PM}$ by employing specific rules to prune the majority of possible plans.

In this paper, in contrast, we propose to employ *partition-aware union grouping* to reduce $\mathcal{PM}$, resulting in fewer plans to consider. The spirit of the solution is to use union operations to merge the data from different sites thereby reducing the plan space.

[10] exact cardinalities are not required, estimations suffice.

Specifically, the method maps $\mathcal{PM} \mapsto \mathcal{PM}^*$, where $\mathcal{PM}^*$ is the *reduced plan matrix* of the *extended planning space*. This gives rise to the following research question: ***How can $\mathcal{PM}$ be reduced, and how can it be done optimally?***

As an example (used in the remainder of this section), consider the simple $\mathcal{PM}$ illustrated in Figure 1 alongside a possible reduction $\mathcal{PM}^*$. The reduced plan matrix $\mathcal{PM}^*$ will always have the same number of columns, but fewer or equal number of rows. In essence, this transform introduces 0s in some of the reduced matrix's cells to limit the number of possible fragments that can be constructed. In other words the general idea is to *reduce* the size of each column individually, i.e., by grouping together sites given a criteria for group *fitness*.

In the remainder of this section, we introduce the novel concept of *fragmented bushy-plans* and proceed with detailing two approaches to reducing the plan matrix $\mathcal{PM}$, inspired from data-analysis, a non-parametric and a parametric method.

Fragmented Bushy Plans Traditionally, a logical query plan is represented as a tree where the non-leaf nodes are algebraic operators while the leaf nodes represent data. As stated in the introduction, the time complexity of a DP optimiser which does not consider disjunctions during the logical planning phase is $\mathcal{O}(3^n)$. In the worst case, between any two joining triple patterns that are fully partitioned on all sites, there would be $n * s$ partition joins. There are an exponential number of possible ways in which a union leaf node in a logical plan can be split into a combination of sub-unions and there are n such union nodes. The higher order exponential space complexity of the *extended planning space* prevents us from exploring all possible plans. Instead, to benefit from parallelism, we consider to explore only a special class of bushy plans, which we call *fragmented bushy plans*.

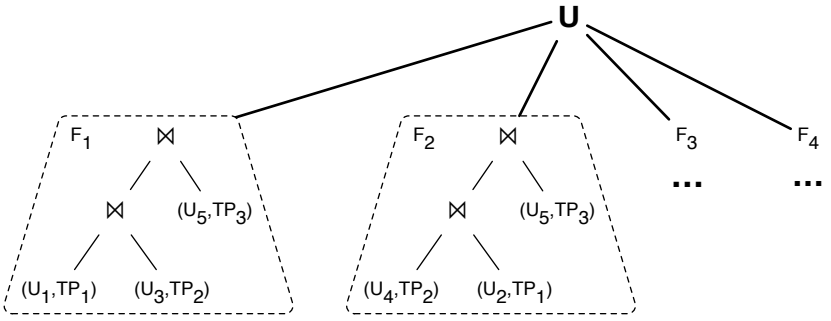


Fig. 2: A possible *fragmented bushy plan* for the example $\mathcal{PM}^*$ from Figure 1. The plan consists of 4 fragments, each equivalent to a left-deep logical plan tree.

Given the construct of a *plan fragment* as outlined in Definition 1, a *fragmented bushy plan* is defined as follows:

Definition 2. *A **fragmented bushy plan** is a logical plan whose root node is a disjunction between multiple **plan fragments**.*

As an example, Figure 2 is a partial illustration of a possible fragmented bushy plan extracted from the reduced plan matrix $\mathcal{PM}^*$ from Figure 1. The primary benefits of fragmented bushy plans are twofold:

- the size of the *reduced extended planning space* (explained by $\mathcal{PM}^*$) can be directly controlled by varying the maximum number of desired unique fragments or ϕ, and
- they are easily parallelisable, since each fragment is independent and can be executed concurrently.

Consequently, in order to explore the new *extended planning space* X-AVALANCHE's optimiser pipeline consists of two general phases:

1. *reduce* the plan matrix $\mathcal{PM}$ and
2. *optimise* each fragment in parallel.

As a result, the optimiser's overall time complexity is $\mathcal{O}(k * n^2 * s^2 + p * 3^n)$, where $p = \frac{\phi}{\#CPU}$ is a constant factor regarding parallelism.

Non-Parametric Optimal Reduction The primary appeal of these methods is that they do not require user-intervention to decide how to best reduce $\mathcal{PM}$. In the remainder of this section, we first detail how a column can be reduced and then show how this method can be used to reduce $\mathcal{PM}$.

A class of methods which can be used to achieve the reduction of each column in $\mathcal{PM}$ are *change point* or *step detection* methods PAGE [1955]. Widely used in statistical analysis, these methods try to identify when the probability distribution of a series of events changes, resulting in a *change-point*. Given this information, the original set of events can be approximated by a piece-wise constant model, a process we refer to as *segmentation*. One such method is *bayesian blocks*, detailed in Scargle et al. [2013], which achieves an optimal reduction of its input (in our case a $\mathcal{PM}$ column) by employing dynamic programming or DP in short. Applied to each column in $\mathcal{PM}$, it features a time complexity of $\mathcal{O}(n * s^2)$ with an $\mathcal{O}(s)$ space complexity, where s is the number of sites and n the number of columns (or triple patterns in query).

While the primary advantage of such methods is that they are parameter agnostic, they do however require ex-ante knowledge about the prior distribution of the data to be segmented. In our case, they require knowledge about the prior probability distribution of triples to participating sites. This can be problematic as data distributions may change, requiring re-learning the prior in order to produce higher quality plans.[11]

The bayesian-blocks algorithm can be used to reduce $\mathcal{PM}$ as seen in Algorithm 7. Iterating over all columns in $\mathcal{PM}$ (line 4), the method retrieves the optimal segments $\mathcal{S}$ for the current column δ (line 5). It than constructs the reduced or segmented column by replacing all cardinalities within each segment $\sigma \in \mathcal{S}$ with their sum (line 6). Other aggregate functions could be used to get better estimates of the size of the resulting union over the given segment. We chose Σ since it represents the upper bound of the estimated cardinality of the union. Finally, the newly reduced columns are concatenated in matrix $\mathcal{PM}^*$, by employing a *full outer join* (line 7).

[11] we used the same prior p_0 as in Scargle et al. [2013].

Algorithm 7 Non-Parametric Plan Matrix Reduction

Precondition: $\mathcal{PM}$: the cardinalities matrix of size $s \times n$, p_0: the prior probability distribution

```
1: function AUTOREDUCE(PM)
2:     cols ← ∅                                          ▷ the columns of PM*
3:     s, n ← SHAPE(PM)                                  ▷ shape: (rows, columns)
4:     for δ ∈ PM do                                     ▷ iterate over all columns ∈ PM
5:         S ← BAYESIANBLOCKS(δ, p0)                     ▷ optimum segmentation of column given p0
6:         cols ← cols ∪ {∑ δ[σ] | ∀σ ∈ S}
7:     return ⋉ cols                                     ▷ outer join of all reduced columns
```

Parametric Optimal Reduction Methods in this class expect the user to pass domain knowledge encoded as parameters. While tedious, this form of *loose coupling* exhibits the major advantage of ease of adaptation when the domain changes or when encoding this knowledge is difficult or expensive. In our case, the domain knowledge is represented by the distribution of cardinalities (or selectivities) of query triple patterns to sites, which is expected to change as data diversifies and its volume increases over time.

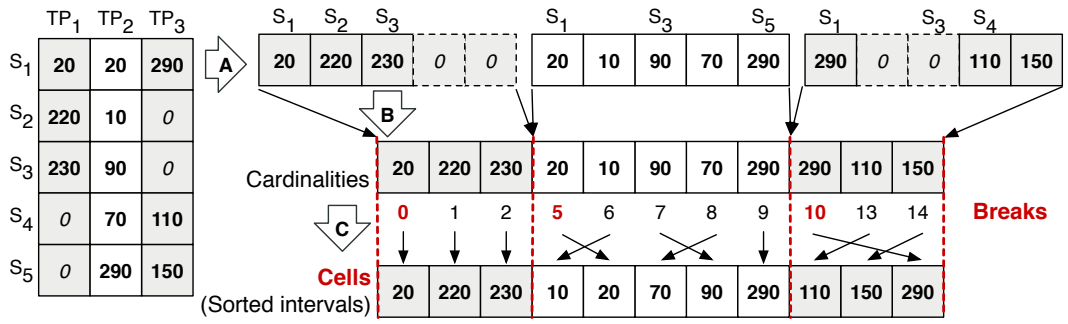


Fig. 3: Preparing $\mathcal{PM}$ for reduction. $\mathcal{D}$ (*cells*) is the array of non-zero cardinalities from $\mathcal{PM}$ in **column major order** form, while B (*breaks*) encodes the position of the columns in $\mathcal{PM}$.

To this end, we adapt the traditional 1D *k-segmentation* method. Unlike the non-parametric bayesian-blocks method which was applied locally to reduce each column in $\mathcal{PM}$, the global parameter ϕ (the number of segments) requires that the method be applied to the entire $\mathcal{PM}$ and not individually to its columns. To achieve this, $\mathcal{PM}$ is represented as an array in *column major order*, i.e., as a 1D array comprised of the concatenation of all columns in the order in which they appear in $\mathcal{PM}$. Figure 3, illustrates the process of representing the plan matrix as a 1D array $\mathcal{D}$ in *column major order* form. In order to avoid the creation of segments that would span across multiple columns and therefore invalidating the semantics of the original SPARQL query, we introduce a helper structure referred to as *breaks*

($\mathcal{B}$), which is a list holding the starting index of each $\mathcal{PM}$ column in $\mathcal{D}$. The structure is used by the DP algorithm to set the cost on any segment spanning over multiple columns to ∞ (line 4 in Algorithm 8).

Algorithm 8 k-Segmentation with Breaks

Precondition: $\mathcal{D}$: numeric array containing data to be segmented, k: desired number of segments, $\mathcal{B}$: integer array with index bounds of non-breakable segments from $\mathcal{D}$

```
1:  function COST(D, j, i, B)
2:      for b ∈ B do
3:          if j < b ≤ i then
4:              return ∞
5:      return MAX(D[j : i]) - MIN(D[j : i])

6:  function KSEGB(k, D, B)
7:      N ← LENGTH(D)
8:      DP ← MATRIX(k,N,∞)          ▷ matrix of size (k, N), elements initialised to ∞ cost
9:      PT ← MATRIX(k,N,0)          ▷ matrix of size (k, N); for solution reconstruction

10:     for 0 ≤ j < k do            ▷ initialisation
11:         DP[j, j] ← 0
12:     for 0 ≤ i < N do
13:         DP[0, i] ← COST(D,0,i,B)
14:     for 1 ≤ j < k do
15:         for j + 1 ≤ i < N do
16:             C ← {DP[j − 1, l] + COST(D, l + 1, i, B) | ∀l ∈ [0, i)}
17:             best ← ARGMIN(C)
18:             DP[j, i] ← C[best]
19:             PT[j, i] ← best
                                    ▷ final solution reconstruction
20:     return SOLUTION(PT, k, N)
```

A DP algorithm, *ksegb* finds the optimal set of segments of $\mathcal{PM}$ – in *column major order* form, $\mathcal{D}$ – with the restriction that any segmentations with segments containing elements from $\mathcal{B}$ are ignored. The algorithm's time complexity is $\mathcal{O}(k * n^2 * s^2)$, where k is the number of segments and a parameter of *ksegb*. The method is exhaustive as it explores all possible segmentations of $\mathcal{D}$. To find the optimal segmentation, the fitness function (line 1) computes the max-min delta of a segment. We base this formulation on the simplifying assumptions that:

1. all unions are executed in parallel, and
2. the time to execute a union is primarily dependent on the selectivity of the given triple pattern.

Therefore, the fitness function of a segment is intended as a measure of *wasted time*. In cardinality-homogenous segments all sites finish around the same time, while heterogeneous segments incur waiting times on sites with lower cardinalities.

The transition from the original parameter ϕ representing the number of plan fragments in $\mathcal{PM}$ to the number of segments required by *ksegb* is performed using the formula from Equation 4.

$$k = n * \sqrt[n]{\phi} \tag{4}$$

Algorithm 9 Parametric Plan Matrix Reduction

Precondition: $\mathcal{PM}$: the cardinalities matrix of size $s \times n$, ϕ: the maximum number of plan fragments to reduce to

1: **function** REDUCE($\mathcal{PM}$, ϕ)
2: $cols \leftarrow \emptyset$ ▷ *the columns of* $\mathcal{PM}^*$
3: $s, n \leftarrow$ SHAPE$(\mathcal{PM})$ ▷ *shape: (rows, columns)*
4: $k \leftarrow n \times \sqrt[n]{\text{MIN}(\phi, s^n)}$
5: $\mathcal{D} \leftarrow$ TOCOLUMNMAJORORDERFORM$(\mathcal{PM})$
6: $\mathcal{B} \leftarrow$ COLUMNPOSITIONS$(\mathcal{PM}, \mathcal{D})$
7: $\mathcal{S} \leftarrow$ KSEGB$(k, \mathcal{D}, \mathcal{B})$ ▷ *optimum k-segmentation*
8: ***for*** $(i, j) \in \mathcal{B}$ ***do*** ▷ *(begin, end) of each column*
9: $\delta \leftarrow \mathcal{D}[i : j]$
10: $cols \leftarrow cols \cup \{\sum \delta[\sigma] \mid \forall \sigma \in \mathcal{S}[i : j]\}$
▷ *outer join of all reduced columns*
11: ***return*** $\bowtie cols$

The parametric reduction method detailed in Algorithm 9 starts by preparing the input for the *ksegb* method. It first represents $\mathcal{PM}$ in *column major order* form (line 5) after computing the number of segments k. It then records the start positions of the original columns in $\mathcal{B}$ (line 6). Next, it obtains the optimum segmentation of the transformed $\mathcal{PM}$ (line 7). Afterwards, it proceeds to constructing the reduced columns, by replacing all cardinalities within each segment $\sigma \in \mathcal{S}[i : j]$ with their sum (line 10), following the same rationale as in Algorithm 7. Finally, the newly reduced columns are concatenated in matrix $\mathcal{PM}^*$, by employing a *full outer join* (line 11).

3.3 Parametric vs. Non-Parametric Fragmentation

Naturally, an automatic or non-parametric reduction of $\mathcal{PM}$ is preferred, given that the optimiser or administrator does not have to be concerned with specifying extra parameters. Such a choice, however, leads to the following question: *how does the non-parametric bayesian blocks method perform in general?* or more specifically, *how does bayesian blocks perform in automatically choosing the number of segments* ϕ*?* To find out the answer to this question, we conducted an exploratory analysis of the bayesian blocks method over synthetically generated data. Hence, we randomly generated synthetic $\mathcal{PM}$ data which simulates the case of a medium-sized 10 triple pattern SPARQL query, with a random distribution of triples to endpoints.

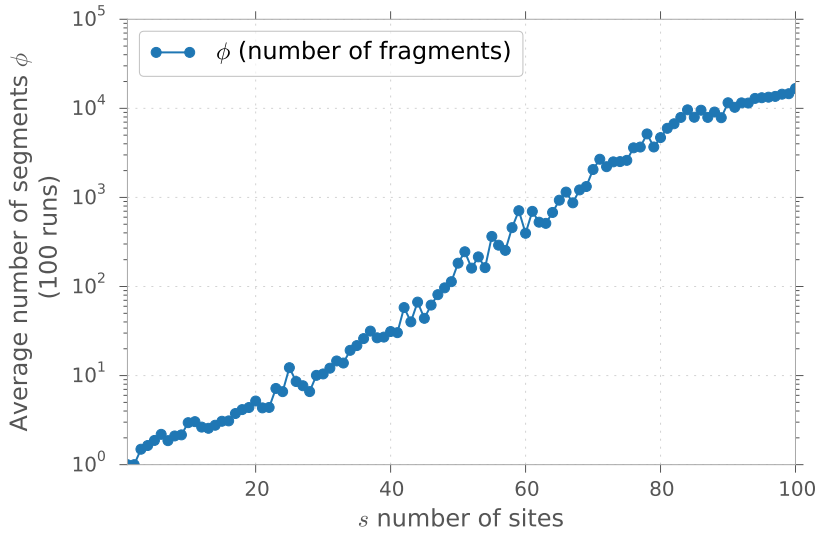


Fig. 4: Number of fragments ϕ automatically selected by *bayesian blocks* function of number of sites s, for a 10 triple pattern query.

Figure 4 shows the relationship between the number of fragments ϕ and the number of sites s, when s increases from a centralised setup to a large federation of 100 SPARQL endpoints. Each datapoint represents the average of 100 runs over randomly generated cardinalities while incrementing the number of sites. A clear observation is that when the number of sites increases over a particular threshold, ≈ 50 for this analysis, the number of fragments automatically chosen by the *bayesian blocks* method starts to increase exponentially. This is undesirable for two reasons:

1. the optimiser cannot choose ϕ in order to control resource wastefulness, and
2. ϕ can, on average, grow very large which diminishes the tractability of the query execution, e.g., more than 10000 fragments for ≈ 90 sites.

In contrast, parametric methods hand control over to the optimiser or the administrator, allowing for the choice of a value that also encompasses resource availability.

Another interesting aspect of the $\mathcal{PM}$ reduction process has to do with with the complex relationship between: (a) the quality (or cost) of the fastest and slowest fragments and (b) the number of chosen fragments ϕ. Figure 5 illustrates this relationship, by comparing the quality of the *fastest* and *slowest* fragments when ϕ is incremented (for parametric methods). We compute the cost o a fragment given the simplifying assumption that query performance is mostly affected by the number of partial results generated by that respective fragment. One can clearly see that the higher the number of fragments, the better the plan fragment quality (lower cost is better). This is no surprise since by fragmenting the original plan, the optimiser ends up dividing the work optimally between participating sites. However, just like before, having a larger number of fragments incurs resource wastefulness leading to a larger overall system load. This

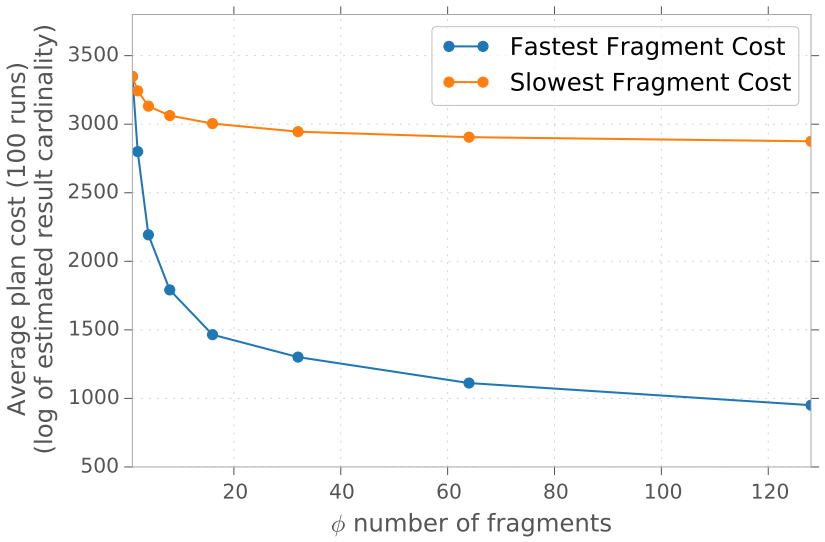


Fig. 5: Quality of plans function of ϕ (maximum number of fragments) for a 10 triple pattern SPARQL query. The cost is equivalent to that of the traditional DP planner when $\phi = 1$.

leads the way to the following questions: *What is an optimal number of fragments ϕ and how can it be computed?*

Results from Figures 4 and 5 show that both methods have their own plusses and minuses. Automatic, non-parametric methods like *bayesian blocks* suffer from the need of precise fine tuning to data. Even in such cases, there is no guarantee that the appropriate number of fragments is low enough for query execution to become tractable. In contrast, parametric methods offer the administrator or query optimiser just this: control over the number of segments. On the down side, finding out the appropriate ϕ can be an expensive trial and error process, considering the complex relationship between the characteristics of each participating SPARQL endpoint and the a fragmented query execution.

Consequently, we chose to employ the parametric *k-segmentation* space reduction method as part of X-AVALANCHE's optimisation, primarily due to its intrinsic control over the number of fragments.

3.4 Total/First Results Tradeoff

Since *fragmented bushy plan* are a variant of bushy plans where the top subtrees represent disjoint partitions or fragments of the query plan, they are easily parallelizable given the fact that each fragment is independent. In consequence, they offer control over executing only a portion of the query if needed. This can be advantageous, for example, in multi-query optimisation situations, when the scheduler can choose to interleave the execution of fragments belonging to different queries in an informed way, avoiding the starvation of clients waiting for results from expensive queries. Another situation where plan fragmentation can be beneficial is when FAST FIRST results constraints are imposed by some application, e.g. a search engine requiring results for the first page.

The fragmented execution of any query plan ultimately offers the caller a tradeoff between t, the time until first results are found and T, the total query completion time. Naturally, minimising both performance metrics is desired. To obtain a clear and quantifiable view of this tradeoff we combine both time measurements within the unified performance metric τ. We express τ using the *euclidean* norm to compute the distance to the ideal, $(0, 0)$:

$$\tau = \sqrt{t^2 + (\delta)^2}, \quad \delta = T - t \tag{5}$$

It is our hypothesis that there exists a number of fragments $\phi > 1$ where the τ performance metric is optimal. Additionally, we expect τ to degrade as fragmentation increases over a given threshold due to the fact that overall system occupancy increases in addition to the overhead and interactions introduced by orchestrating the execution of a large number of fragments.

4 Scalable Distributed Unions

When data pertinent to a triple-pattern or subquery is physically partitioned among several sites, the optimiser will have to consider a disjunction between all relevant endpoints in order to guarantee result-set completeness. To simplify matters, most state of the art query optimisers will not consider different grouping strategies during the logical planning phase. Consequently, unions are only applied to the relevant leaf nodes of the plan. If the number of endpoints is large, the physical design of the operator can have a dramatic effect on the overall query execution performance. Consider for example a setup similar to the one illustrated in Figures 1 and 2, with the difference that instead of 5 sites data is partitioned over 100 sites. Such a scenario could lead to unioning triples matching for example TP_1, from 100 endpoints.

While there are many ways in which a distributed union can be carried out, in the following we will focus on methods where computation occurs remotely and not at the client site. Specifically, we investigate parallel execution while considering the naïve serial method as the baseline. The main advantage of the serial method lies in its inherent simplicity: it does not require advanced support, aside from the basic assumption that *2-way unions* can be carried out by simply *shipping* the smaller result set of bindings to the target server and performing the union in-place. Obviously, this serial approach forgoes any performance benefits from parallelism since unioning n sites require in the order of $\mathcal{O}(n)$ union operations.

On the other spectrum from serial execution all binding sets or partial results can be shipped to a previously elected *master* site and 'unioned' in-place. In this case, since all union operations are executed in parallel, the cost of the union falls in the $\mathcal{O}(1)$ complexity class and would theoretically be equivalent with the cost of the most expensive of the union operations. In practice, the *master* site can become a bottleneck when there are many sites, by having to keep n remote connections open at the same time. Furthermore, if duplicate partial bindings are dropped either to reduce traffic or

due to the UNIQUE modifier, local contention can mitigate the benefit of parallelism and would require more complex handling strategies that are not implemented by most federated RDF stores.

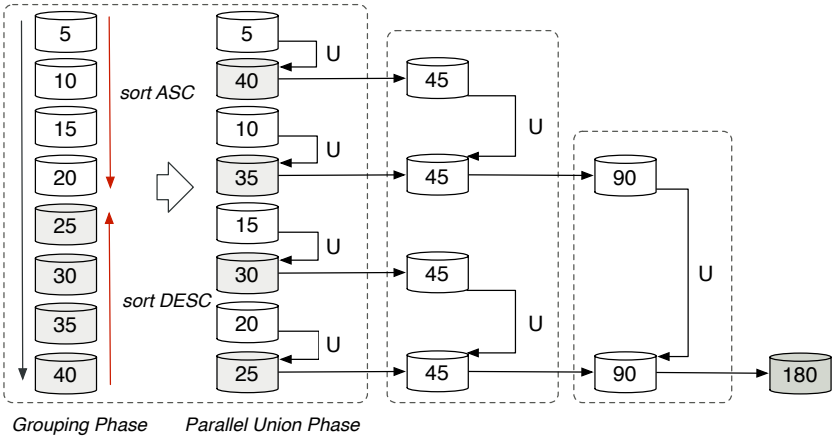


Fig. 6: Example *parallel tree union* for 8 sites. Numbers are subquery cardinalities on each site.

Algorithm 10 parallel tree union

Precondition: $\mathcal{S}$: the participating sites, given subquery sq

```
1: function parallelUnion(sq, S)
2:     sortAsc(S)                          ▷ sort S on cardinality (ascending)
3:     while ¬empty(S) do
4:         n ← |S|/2
5:         slaves ← S[: n]                 ▷ left side of union
6:         masters ← S[n :]                ▷ right side of union
7:         sortDesc(masters)               ▷ sort master sites on cardinality (descending)
                                           ▷ parallel execution of each (slave,master) union pair
8:         S ← parmap({s ∪ m | ∀s ∈ slaves, m ∈ masters})
9:     return S[0]                         ▷ the root of the union-tree holds all partial results
```

In the following, we propose a simpler distributed union execution strategy which enjoys both: the benefit of parallelism while at the same time requiring only the simple *2-way union* capability from a participating site. Called *parallel tree-union*, the method uses the topology of a balanced binary tree with endpoints as nodes. The algorithm traverses the tree bottom-up towards the *master* endpoint, by iteratively pairwise unioning each level of leaf nodes. The time complexity for this operator is $\mathcal{O}(log(n))$ for n sites.

As illustrated in Figure 6, within each iteration the sites are divided into two groups: *slave* and *master* sites, where the latter are the ones performing the union. To load-

balance the amount of traffic that is generated, the slave with the smallest binding-set ships to the master with the most partial bindings in each iteration (lines 5 - 8 in Algorithm 10). This has the advantage of producing more balanced later stage unions.

5 Distributed State Management & Caching

One of the major factors contributing to X-AVALANCHE's increased performance is the distributed management of partial query results. The SPARQL 1.1 federation extensions are stateless and therefore operate at a lower level than X-AVALANCHE. They are however instrumental building blocks, since X-AVALANCHE relies on: *i*) the COUNT aggregate, to obtain statistics about triple patterns (equivalent statistics can be retrieved using W3C's VoID), *ii*) the SERVICE keyword, to execute a subquery agains a remote endpoint, and *iii*) the VALUES clause, to constrain the results another endpoint. During execution, network traffic is minimised, by keeping materialised BGP views in memory for the duration of the current query as detailed in Başca and Bernstein [2014].

Parallel Multicast Joins The introduction of support for disjunctions triggered the addition of support for the execution of parallel joins. Each X-AVALANCHE endpoint can multicast and coordinate a join operation between multiple remote endpoints. All join operations are *bind semi-joins*, where a set of partial bindings is shipped remotely to reduce the execution of the subquery using the SPARQL 1.1 VALUES clause. Consider for example the case of the star query $LQ5$ (Listing 6.5), where during the execution process bindings for the `?name` variable are restricted to two values: *"GraduateStudent1"* and *"GraduateStudent2"*. Hence, the execution of the remainder triple patterns is bounded on all relevant remote sites of the semi-join, by the two values.

In addition, the source partial results table from which the bindings for the join variable are shipped, can be reconciled using either a *bloom filter* of the remote set of partial bindings if the set is large, or the (compressed) set otherwise. Consider for example that the remote side, or destination, of the semi-join operation produces partial results only for the *"GraduateStudent1"* binding of the `?name` variable. Therefore all partial records from the source endpoint matching *"GraduateStudent2"* can be safely discarded.

Execution by Proxy In addition, just like p2p systems, all X-AVALANCHE operators can be executed directly or by proxy. Proxy based execution helps the endpoint orchestrating the overall query execution to offload part of the execution orchestration to remote sites while still managing the overall process. This is a particularly useful design since it allows an RDF federation engine more flexible management of remote resources and significantly aids in introducing more parallelism into the query execution pipeline.

For example, consider the *bind semi-join* operation for query $LQ5$ on variable `?name` described earlier. The query execution coordinator can manage the process in two ways. It could orchestrate the process directly by managing each of the phases of the semi-join operation, or it could delegate the management of the entire semi-join operation to the designated source endpoint, therefore benefiting from more I/O and computational resources to coordinate the execution of potentially other concurrent operations.

SPARQL Endpoint Caching Often the performance of the underlying SPARQL endpoint has a negative impact over the RDF federation engine. To mitigate some of the performance penalties incurred, in X-AVALANCHE we enhance the wrapped SPARQL endpoint with a simple cache. The cache cannot be used to store the results of all SPARQL query types. Cacheable queries include: COUNT queries and simple SELECT queries that do not have a VALUES clause. For obvious reasons, queries which contain VALUES variable binding sets cannot be cached. In such cases the key would have to uniquely identify not only the BGP or subquery but also the supplied binding sets. Creating a unique key in this case can be expensive for large binding sets. We employed a typical LRU cache eviction strategy with expiration for records. In practice, the expiration duration should not be larger than the endpoint's dataset update frequency.

6 Evaluation

In this section we present and discuss the results we obtained from evaluating X-AVALANCHE in a controlled setup in order to observe the impact that different external and internal factors have on system performance. Specifically, we first investigate the impact of the parallel union operator followed by a enquiry of the impact of SPARQL endpoint caching. We then explore X-AVALANCHE's performance whilst varying problem size and data distribution. Additionally, we evaluate X-AVALANCHE against the current top performing federated SPARQL engine: FedX, as identified in Saleem et al. [2014].

Technical setup: We used the latest freely available version of FedX, v3.1.[12] All experiments were run on a cluster of 11 machines, each having 128 GB RAM and two E5-2680v2 @2.8GHz processors, with 10 cores per processor, i.e., equivalent to 20 execution units when *HyperThreading* enabled. Nodes run 64 bit linux (kernel version 3.2.0) and are interconnected using standard 1Gb ethernet. We used Python 2.7.8 and all SPARQL endpoints were powered by Virtuoso v7.1 open source.

6.1 Benchmark Design

X-AVALANCHE is designed to improve query performance in large federations of SPARQL endpoints. However, as mentioned in Section 1.1, the present day LoD's schema richness and broad semantic diversity create a *semantically selective* benchmarking setup, i.e., where the vocabularies used in the query restrict the execution to a handful of endpoints. To address this notion, we distinguish between the selectivity of a query based on the number of result tuples, which we call *result-set selectivity*, and the *source selectivity* of the query. The latter kind of selectivity is the decisive factor during the *source selection* phase and can dramatically improve performance and recall.

Unfortunately, semantically selective benchmarks do not shed any light into how the federation engine performs in worst case scenarios, where hundreds of endpoints are actively engaged in query answering. These scenarios can occur when: *a*) large

[12] http://www.fluidops.com/

numbers of sites operate within the same domain, a clear future development as the LoD continues to grow and *b*) the query is *re-written* to use different but similar schemas (i.e., overlapping semantics). In both situations the federated engine has to coordinate the query execution over a large number of endpoints.

In order to observe X-AVALANCHE's performance improvements compared to state of the art federated engines as well as to better understand the impact of internal (configuration) and external (data distribution / workload) factors in a large federation setting, the benchmark must be able to:

1. scale to as many endpoints as required,
2. allow for data distribution control,
3. emulate a *semantically homogenous* setup over a large number of endpoints,
4. provide a diverse and comprehensive set of queries.

The most comprehensive federated SPARQL benchmark to date is FedBench Schmidt et al. [2011]. It features a mix of synthetic and real-world LoD data. In addition, it offers a set of cross-domain and domain-specific queries. While highly useful, it does not adhere to the above requirements. First concerning *points 4* and *1*, it provides only a fixed data set size, while queries do not systematically cover a defined design space. Second, *point 3* is not addressed, as it, for example has only three sources for life sciences queries. Finally, regarding *point 2*, the data distribution is not specified.

To mimic this worst-case scenario, we modified the popular LUBM Guo et al. [2005] benchmark to generate data from a single domain: academia. Both scale and data distribution are user controllable. While there are many possible data distributions, in our evaluation we adopted *horizontal partitioning*. Highly popular, these strategies often provide an excellent tradeoff between performance and ease of use. For example, Huang et. al. Huang et al. [2011] show substantial performance improvements by employing a partitioning scheme based on the idea that *star shaped* queries are common and therefore star shaped sub-graphs should not be split. Finally, horizontal partitioning schemes are a natural fit for federations of RDF data, as it is unlikely for triples to be randomly assigned to sites that belong to different administrative entities, but very likely for triples sharing a common provenance criteria to stay together.

Given its popularity, we adopt this partitioning scheme and choose 5 horizontal splits with increasing levels of *distribution messiness*. In the first distribution $U1$, data specific to one LUBM university is allocated to one site, similarly, distributions $U3$, $U5$ and $U7$ split the triples of each university to 3, 5 and 7 sites respectively.[13] Finally, we complemented these with distribution UH which represents the traditional horizontal split of the data based on the *subject* of a triple. While UH is not a natural fit for federated setups it offers valuable insight.

Accompanying these distributions we developed 14 SPARQL queries (cf. detailed in Appendix B and Table 1) with different shapes as specified by the Waterloo SPARQL

[13] We released the LUBM generator wrapper under an open source licence at https://github.com/cosminbasca/rdftools

Table 1: federated LUBM queries

Query	Shape	Selectivity	Scaling
$LQ1$	LINEAR	LOW	SCALING
$LQ2$	LINEAR	HIGH	CONSTANT
$LQ3$	LINEAR	HIGH	CONSTANT
$LQ4$	LINEAR	LOW	SCALING
$LQ5$	STAR	HIGH	CONSTANT
$LQ6$	STAR	LOW	SCALING
$LQ7$	STAR	LOW	SCALING
$LQ8$	STAR	LOW	SCALING
$LQ9$	FLAKE	HIGH	CONSTANT
$LQ10$	FLAKE	LOW	CONSTANT
$LQ11$	FLAKE	HIGH	CONSTANT
$LQ12$	COMPLEX	HIGH	CONSTANT
$LQ13$	COMPLEX	HIGH	CONSTANT
$LQ14$	COMPLEX	LOW	CONSTANT

Diversity Test Suite (WatDiv) Aluç et al. [2014].[14] In addition to shape, the queries are also split into *high* and *low* result-set selectivity given a threshold on the total number of result tuples. Furthermore, we differentiate between *constant* and *scaling* queries when their result sets stay constant or increase with total dataset size. For this evaluation we fixed the result-set selectivity threshold to **5000** tuples.

6.2 Union Operator Performance and Scaling

To ascertain how much faster the parallel tree union operator is when compared to the baseline serial union we constructed four queries each containing only a single triple-pattern: $LU1$ - $LU4$ (see Appendix C). Note that these single triple pattern queries have the advantage over $LQ1 - LQ14$ that they solely measure the impact of different kinds of unions. Specifically, we measured the time it takes to union all partial bindings spread over 100 sites, while relying on the same experimental setup detailed earlier.

Table 2: Union LUBM queries

Query	Cardinality	T_{SERIAL}	$T_{PARALLEL}$
$LU1$	19	0.26	**0.20**
$LU2$	8000	6.48	**1.02**
$LU3$	25600	7.09	**1.23**
$LU4$	160006	10.27	**1.97**

* T: time for all results (seconds)

[14] The queries are also available publicly athttps://github.com/cosminbasca/rdftools/blob/master/doc/DESCRIPTION.md

As seen in Table 2, the final result-set cardinality for each of the union-queries varies between ≈ 20 and 160000 tuples. As expected, when the cardinality of the result set is low the methods fare comparably in terms of performance. For example query $LU1$ produces only 19 result tuples, a much smaller number than the number of participating sites. In consequence, and assuming no data replication in our setup, not all endpoints can contribute to the final result. This leads to a low number of disjunctions for both operators, and hence similar performance: ≈ 0.2 seconds.

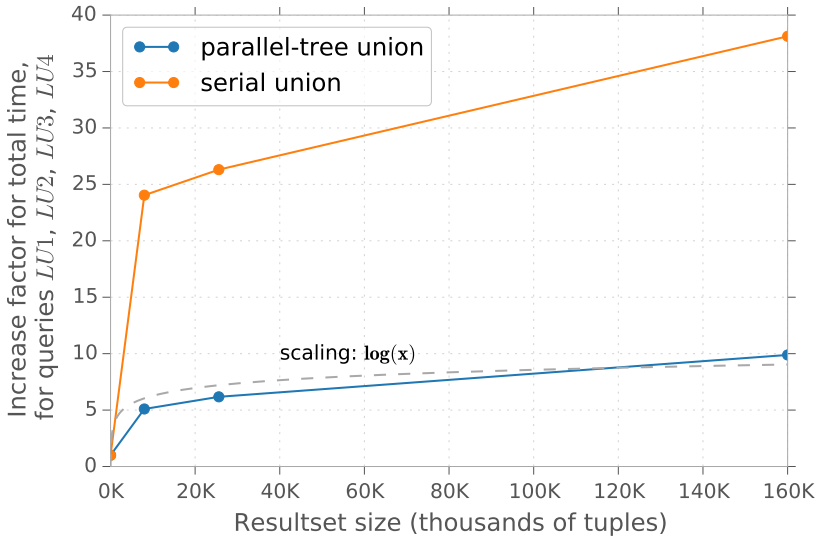


Fig. 7: Parallel tree vs serial union performance function of varying triple-pattern cardinality. Partial bindings horizontally partitioned over 100 sites.

However, when more data is involved i.e., for queries which produce more partial results, the performance difference can be dramatic. Just as expected (and graphed in Figure 7), the parallel tree union operator exhibits a scaling characteristic closely following a logarithmic performance degradation (blue versus dashed line). It is however interesting to observe that the naïve serial operator also scales better than linear when result-set cardinality increases. This can be explained by the fact that larger result-sets use the network more efficiently, by saturating bandwidth, unlike smaller result-sets which do not utilise the entire available bandwidth.

As seen, for queries $LU2$, $LU3$ and $LU4$, the parallel tree union algorithm leads to a **6.3x**, **5.7x** and **5.2x** performance boost over the naïve serial case. Even more so, such performance gains are typically cumulative, since even simpler queries may require several union operations – proportional with the number of (partitioned) triple patterns in the query. Additionally, the same general principle can be applied not only to union but to merge operations as well.

6.3 Impact of SPARQL Endpoint Caching

In order to observe the extent by which the SPARQL endpoint caching strategy (outlined in Section 5) impacts overall system performance, we measured the geometric mean over the entire benchmark, of the time spent while waiting for RDF store results. We differentiated between the two optimisers employed by x-Avalanche. Furthermore, we used the same experimental setup detailed before and controlled for fragmentation by setting the number of fragments $\phi = 1$.

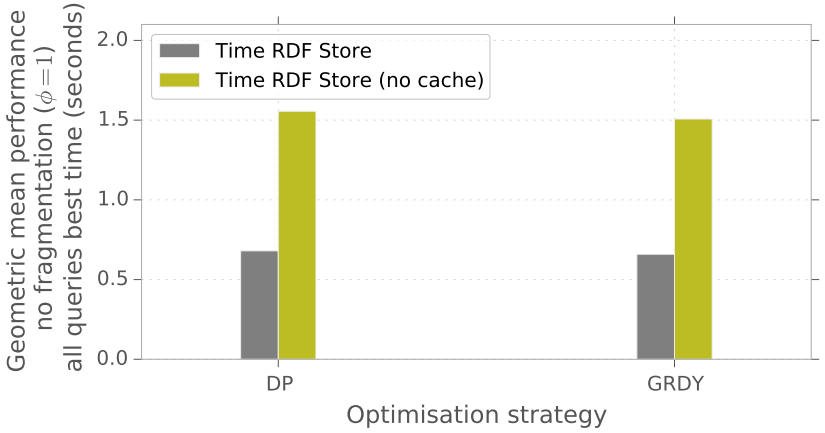


Fig. 8: SPARQL endpoint cache impact.

Figure 8 graphs the geometric mean of the SPARQL endpoint wait times incurred (i.e., the time that x-Avalanche waits for results) for all queries in the benchmark. As can be seen, caching has a significant impact on overall performance. The hit ratio varies from **52%** to **66%** with an average of 55% cache hits. The impact is significant even for high performance SPARQL endpoints, like Virtuoso v7.1 (used in this evaluation), and resulted in an $\approx$ **10%** reduction of the benchmark overall geometric mean query completion time.

Note that these results are based on the simple strategies that only cache the results of BGP and COUNT queries. More elaborate strategies are likely to have a higher impact.

6.4 System Scalability

Performance scalability is critical to any distributed DBMS query processing engine when more data is indexed. To this end, we varied the size of the generated LUBM datasets from 500 universities totalling $\approx$ 67 million triples to 8000 universities totalling $\approx$ 1.1 billion triples. Naturally, the scaling characteristic of the underlying SPARQL endpoint, which x-Avalanche wraps, has an impact on parts of the federated engine's execution pipeline. In the worst case, if all endpoints would be powered by an RDF store that scales poorly, the maximum number of SPARQL operations that need to be executed serially, i.e., the *critical execution path*, will be the primary performance

impacting factor. X-AVALANCHE mitigates the effect of a low performing RDF store to a certain degree by: *1*) caching the results of queries without VALUES bindings (Section 5) and *2*) keeping materialised views in memory for the current executing query.

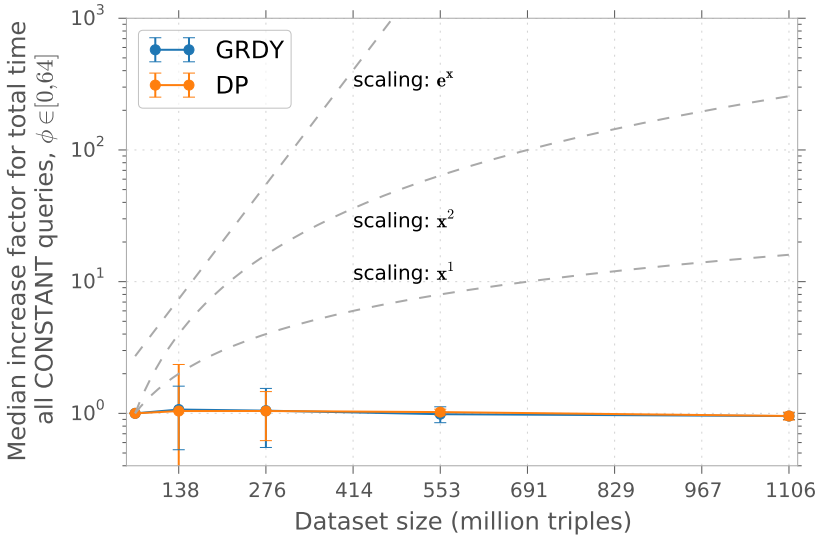


Fig. 9: Performance scaling by strategy, when dataset size increases for *constant* queries (error bars indicate standard deviation).

First, we examine how X-AVALANCHE's performance scales when processing queries whose number of results do not change when the total number of triples stored across all endpoints varies. Figure 9 graphs the median ratio between the total query time across all *constant* benchmark queries for the current dataset size and the smallest dataset: LUBM 500. As observed, X-AVALANCHE exhibits average constant scaling for queries whose number of results stay the same at all dataset sizes i.e., **is unaffected by dataset size variation for constant queries**.

Similarly, we examine how X-AVALANCHE's performance scales when dealing with queries whose number of results increase with the dataset size. In our evaluation queries from the *scaling* group (see Table 1) exhibit the same scaling factor as that of the dataset, e.g., if the dataset size doubles so does the number of results for the respective query. As can be observed in Figure 10, X-AVALANCHE's median scaling characteristic is better than the theoretical *linear scalability* threshold (bottom most dotted line in Figure labeled: *scaling:* x^1). While there are cases that lead to performance degradation as dataset size increases (error bars in Figure) they still follow a linear scaling characteristic as depicted. In conclusion, X-AVALANCHE **exhibits better than linear performance scaling for scaling queries**.

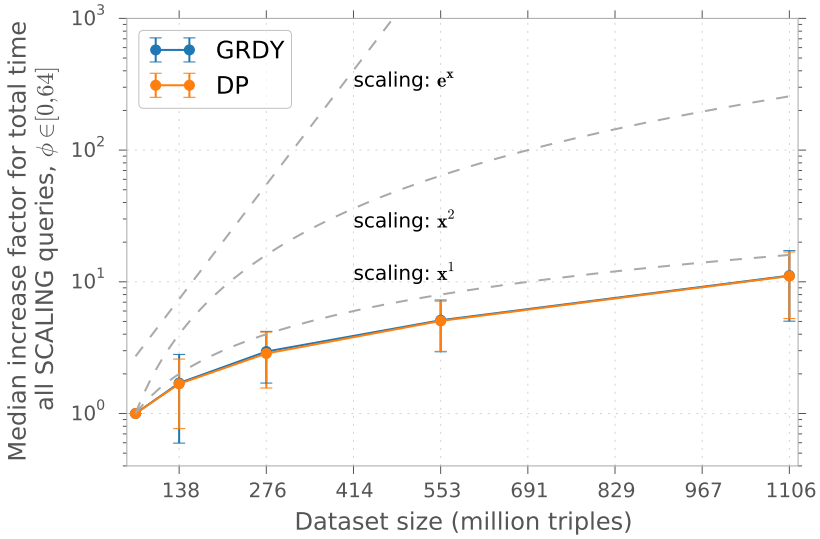


Fig. 10: Performance scaling by strategy, when dataset size increases for *scaling* queries (error bars indicate standard deviation).

6.5 Data Distribution

To see how the X-AVALANCHE optimisation strategies are impacted by varying distribution messiness, we generated distributions $U1$, $U3$, $U5$, $U7$ and UH for the LUBM 8000 scaling factor, a dataset totalling more than 1.1 billion triples.

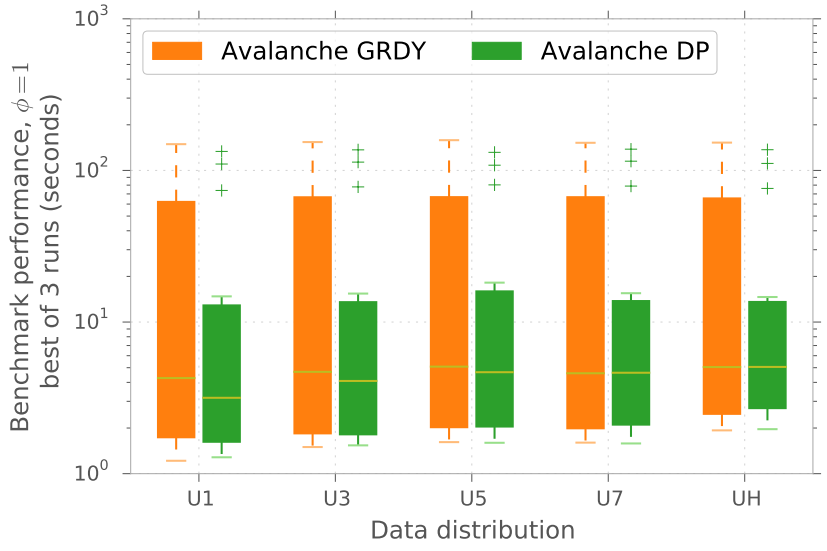


Fig. 11: Overall benchmark performance function of data distribution. The *boxplot* graphs the quartiles (box), the largest and smallest non-outliers (little T-shaped extensions), as well as possible outliers (crosses).

Figure 11 illustrates X-AVALANCHE's top performance distribution over all benchmark queries by optimisation strategy. We controlled for the effects of fragmentation and disabled it by setting $\phi = 1$. Each query was run 3 times and the best run time was considered. Results show that both the greedy (GRDY) and dynamic programming (DP) optimisers exhibit a comparable average performance as a university's triples spread further from the source, i.e., distributions $U1 \longrightarrow UH$. However, as expected the optimal DP optimiser fares better in general than GRDY. The average best performance ranges in the $[3.1, 5]$ and $[4.2, 5]$ seconds intervals for the DP and GRDY respectively. An interesting observation is that while for the messiest distribution UH both show the same average performance, DP is 1 second faster on average for the less messy distribution $U1$.

Performance differences become quite visible however by the time 75% of the benchmark queries have executed. The GRDY optimiser exhibits an average performance between 62 and 66 seconds depending on distribution while the DP optimiser takes only between 12.8 and 15.9 seconds to achieve the same. The DP optimiser is on average ≈ 4.5 times faster than the GRDY approach.

In order to measure the effect of distribution variation on X-AVALANCHE, i.e., to see how robust X-AVALANCHE is to distribution change, we performed *pairwise t-tests*[15] between the obtained measurements of all distribution pairs. We use the more rigorous *three σ rule*, i.e., having a P value threshold of 0.001, to determine if the observed effect is due to distribution variation and not due to chance alone. Distribution variation has no effect on the GRDY optimiser. The P value ranges from 0.003 to 0.93. The smaller P values are obtained when comparing distribution $U1$ to any other distribution. A similar conclusion can be drawn for the DP optimiser with one exception, the effect that distribution $U1$ has on X-AVALANCHE compared to distribution UH is statistically significant with $P = 0.0004$. In conclusion we can safely say that even though performance degrades slightly, the DP optimiser is robust to distribution variation as triples are spread further from the source with the exception of the less messy distribution $U1$, where performance is best for both optimisers.

A similar trend can be seen in Figure 12 which plots the geometric performance of both strategies in addition to FedX for the entire benchmark. Again, while the distribution has a general but limited impact on performance, except for $U1$ where as expected it performs best, it does not affect the relative differences between the two optimisation strategies, with the greedy optimiser consistently performing worst.

6.6 Versus State of the Art

In order to get a better grasp of X-AVALANCHE's performance gains, this section compares its results against the top performing state of the art SPARQL federated engine, which supports *location transparency*. We chose to evaluate only against FedX, since a recent fine grained and comprehensive study Saleem et al. [2014] found that overall FedX outperformed all other state of the art SPARQL federation engines.

[15] We tested for normality using the SciPy normality test, which is based on D'Agostino and Pearson.

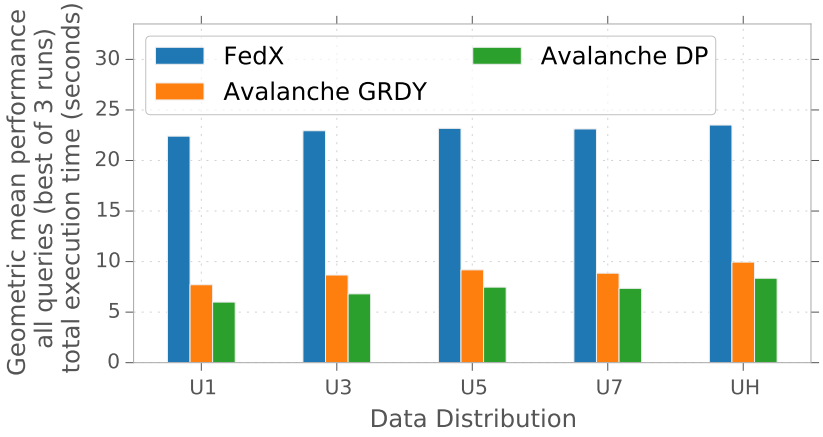


Fig. 12: Geometric benchmark performance function of data distribution, for both systems.

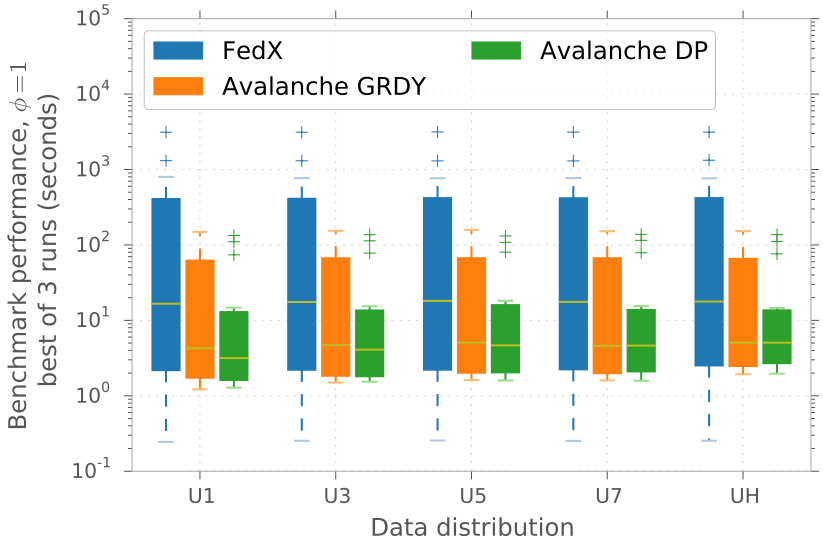


Fig. 13: Overall benchmark performance function of data distribution, for both systems. Note that y-axis is logarithmic.

As seen in Figure 12, both optimisation strategies outperform FedX in the geometric mean over the entire benchmark. Like before, we controlled for fragmentation by setting $\phi = 1$, and considered the best out of 3 runs for each query. Both federated engines perform better for less messy distributions, where the triples are spread out on fewer or no extra endpoints. A detailed statistical breakdown of the performance difference between X-AVALANCHE and FedX is illustrated in Figure 13. Results clearly show that X-AVALANCHE is **more than one order of magnitude faster** than FedX when comparing peak performance for the most expensive query for each system. In addition it is interesting to observe that:

- the slowest X-AVALANCHE query finishes well before FedX completes the benchmark's 75^{th} percentile,
- the 75^{th} percentile of the benchmark queries are completed by the DP optimiser before FedX completes the benchmarks 50^{th} percentile, and
- most expensive non-outlier query for GRDY is comparable with the outlier queries for DP. Furthermore, DP completes the benchmark's 75^{th} percentile significantly sooner than GRDY.

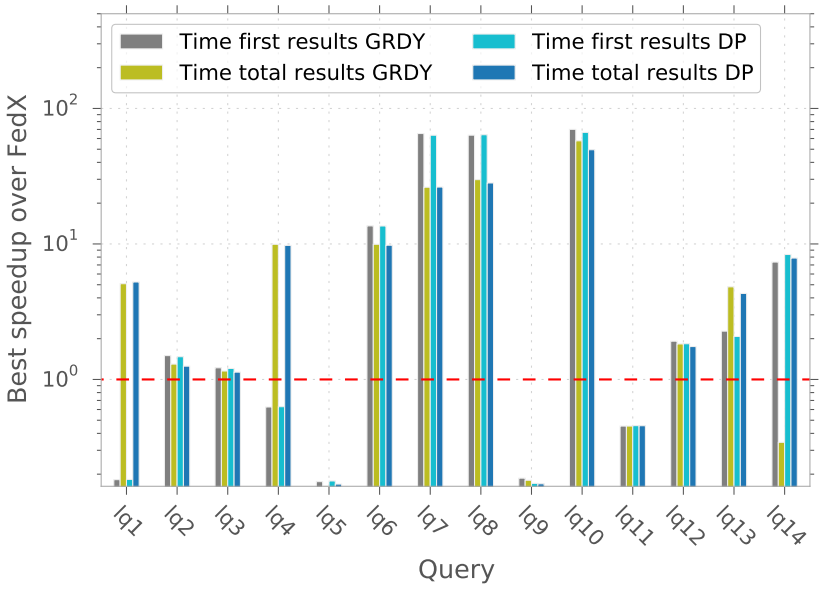


Fig. 14: Best speedup for $U7$, any configuration.

Speedup Figure 14 illustrates X-AVALANCHE's speedup over FedX, by optimiser strategy. In this setup we consider the messiest natural distribution we evaluated namely $U7$, where the triples of a university are spread to 7 other endpoints. We did not control for fragmentation and choose the best response time for per system and per query. The dotted line in Figure represents the *point of equal best performance* between the two systems.

The GRDY optimiser obtains a maximum performance **speedup factor of 70x respectively 57.7x over FedX** for first results retrieval respectively total query completion. While better performing in general the DP optimiser obtains a maximum **speedup factor of 66.5x respectively 49.5x over FedX** for first results respectively total time.

GRDY is faster than FedX for total query performance in 10 of the 14 queries while DP performs better for 11 queries. For getting first results both optimisers are faster for 9 of the 14 queries. FedX is faster in 3 respectively 4 out of the 14 queries over the DP respectively GRDY optimisers, and in 5 queries when retrieving first results. However, as seen in Table 3, the difference between the two systems for queries where

FedX is faster is between 1.1 and 1.7 seconds with the notable exception of query $LQ14$ where GRDY completes in 140.5 seconds compared to 48.2 seconds for FedX. This is an expected result since the query is part of the COMPLEX group and the greedy optimiser does not guarantee optimality. The DP optimiser however, finishes query execution in 6.1 seconds, an expected conclusion. We attribute FedX's speedup over X-AVALANCHE for queries $LQ5$, $LQ9$ and $LQ11$ to the following:

a) the queries are highly selective with 7, 3, and 133 results respectively, and
b) FedX's local cache, which can greatly improve performance by discarding sources known not to contribute to the current query.

Table 3: Best query performance for each system

Query	t_{AVA}^{GRDY}	t_{AVA}^{DP}	t_{FEDX}	T_{AVA}^{GRDY}	T_{AVA}^{DP}	T_{FEDX}
lq1	2.3	2.3	0.4	**4.3**	**4.2**	**22.1**
lq2	1.5	1.5	2.2	**1.8**	**1.8**	**2.3**
lq3	1.6	1.6	1.9	**1.9**	**2.0**	**2.2**
lq4	46.7	46.5	29.3	**77.4**	**78.9**	**769.4**
lq5	1.4	1.4	0.2	**1.6**	**1.5**	**0.3**
lq6	35.7	35.7	485.0	**130.7**	**132.8**	**1299.3**
lq7	5.9	6.0	382.3	**15.5**	**15.5**	**407.5**
lq8	45.3	44.9	2873.7	**105.0**	**111.1**	**3130.8**
lq9	1.8	1.9	0.3	**1.9**	**2.0**	**0.3**
lq10	4.5	4.7	313.7	**7.3**	**8.4**	**418.7**
lq11	2.0	2.0	0.9	**2.0**	**2.0**	**0.9**
lq12	2.3	2.4	4.4	**2.3**	**2.4**	**4.2**
lq13	2.6	2.8	5.8	**2.7**	**3.0**	**13.1**
lq14	5.6	5.0	41.5	**140.5**	**6.1**	**48.2**

* t: time for first results (seconds)
+ T: time for all results (seconds)

For cases where X-AVALANCHE is faster than FedX, the performance difference ranges from near similar, e.g., 0.5 seconds, to **dramatic improvements of over 3000 seconds**, as observed for query $LQ8$ a low selectivity start shaped query with more than 70000 results.

6.7 Fragmentation

In Section 3.4 we introduced the τ performance metric as defined in equation 5. It offers a unified measure of the tradeoff between time to first results and total query execution time. Considering the τ metric, in the following, we investigate X-AVALANCHE's average benchmark performance when plan fragmentation is considered. We varied the number of fragments $\phi \in [0, 64]$ by powers of 2 increment.

In the following we investigate the average benchmark performance of X-AVALANCHE in terms of the τ performance tradeoff.

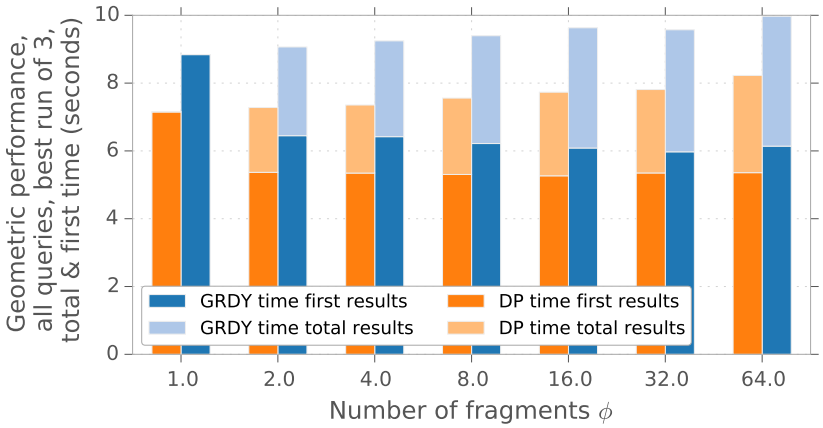


Fig. 15: Overall benchmark performance by number of fragments and optimisation strategy.

Figure 15 depicts the geometric benchmark performance split by total query completion time and time to first results for both optimisation strategies function of number of fragments ϕ. We varied $\phi \in [1, 64]$ by powers of 2 increments. All generated fragments were executed concurrently in parallel on the orchestrating node's 10 physical cores. In general we can see that fragmentation helps deliver FAST FIRST results at the cost of introducing a small penalty for overall completion time. The trend appears to be more accentuated on average for the GRDY optimiser.

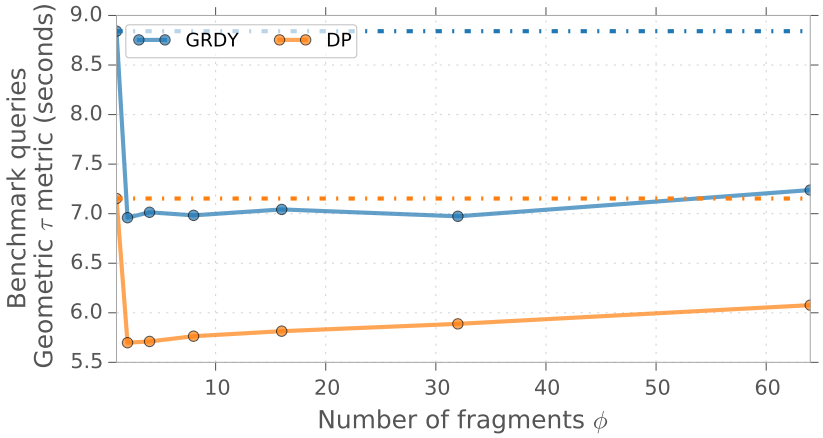


Fig. 16: Overall benchmark performance by number of fragments and optimisation strategy.

A quantifiable view of the trade-off between total and first results is graphed in Figure 16. The dashed line represents τ when $\phi = 1$, equivalent to total execution time when fragmentation is disabled. Both GRDY and DP strategies benefit from fragmentation

if the desired goal is to get first results fast with some penalty in increasing query execution time. For the give experimental setup, an optimal tradeoff is obtained for DP when $\phi \in [2, 4]$, while for GRDY when $\phi \in [2, 32]$. It is interesting to note that on average the greedy approach can offer a better tradeoff when fragmentation is enabled than DP with no fragmentation.

6.8 Query Shape and Selectivity

To get a clearer view of the impact of workload on performance, in the following we control for query shape and result set selectivity.

High selectivity queries They are primarily characterised by low number of results. We consider a query to be highly selective if it has ≤ 5000 results. Consequently, such queries are expected to have better execution performance, leading to the hypothesis that the impact of any optimisation is less visible than for low selectivity queries. This fact is easily observed in Figures 17 through 20, where the range of the τ metric is between 1.4 and 2.75 seconds overall.

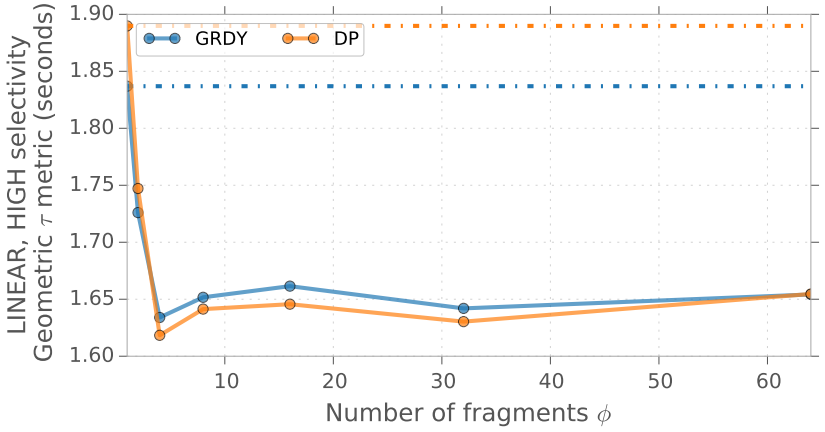


Fig. 17: High selectivity LINEAR queries

For LINEAR, STAR and COMPLEX queries there exists an optimal tradeoff for $\phi > 1$. In general the DP optimiser fares better, however, this is not the case for COMPLEX queries where GRDY offers better performance (Figure 20) although by a very small margin of 0.1 seconds on average. It is interesting to observe that for FLAKE queries (Figure 19), only the DP optimiser benefits from fragmentation with an optimal $\phi \in [2, 4]$ seconds. At the same time the GRDY optimiser shows a steady degradation characteristic although by a very small margin of 0.15 seconds on average.

Low selectivity queries Low selectivity queries are naturally more expensive since they usually produce a larger number of partial results during execution. Therefore, the effects of fragmentation on the different optimisation strategies and query shapes is more visible,

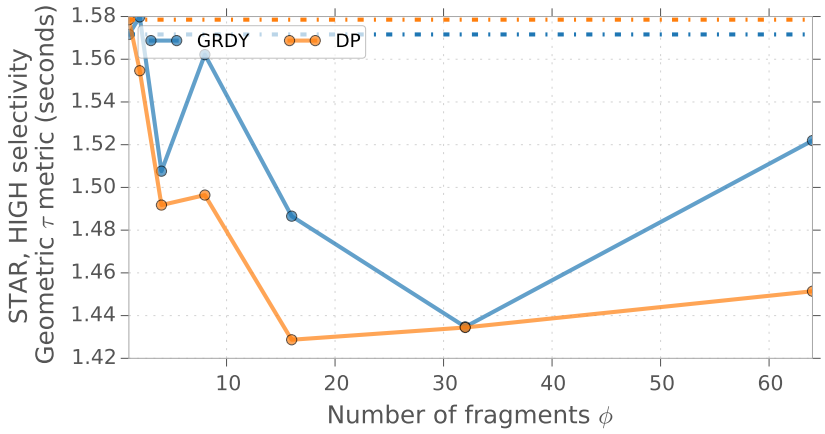


Fig. 18: High selectivity STAR queries

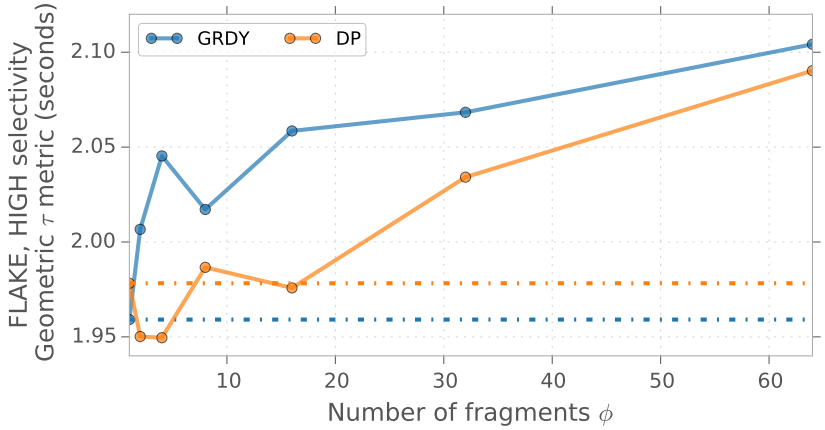


Fig. 19: High selectivity FLAKE queries

as seen in Figures 21 through 24, where the range of the τ metric is between 5.5 and $\approx$ 160 seconds overall.

LINEAR shaped queries show a clear benefit (Figure 21) when fragmentation is enabled. While both optimisers fare similarly in performance, an optimal tradeoff is obtained when $\phi \in [32, 64]$. We believe this to be due to the fact that when fragmented this class of queries leads to less interactions between executing fragments than in other situations.

For STAR shaped queries, both the GRDY and DP optimisers benefit from fragmentation with an optimum trade-off when $\phi = 2$. It is interesting to note that both strategies follow a similar performance trend with GRDY being faster by up to 5 seconds on average. In addition more than 16 fragments leads to performance degradations in our experimental setup.

Perhaps the most dramatic performance improvements are observed for FLAKE shaped queries which benefit both optimisers for any number of fragments in the chosen

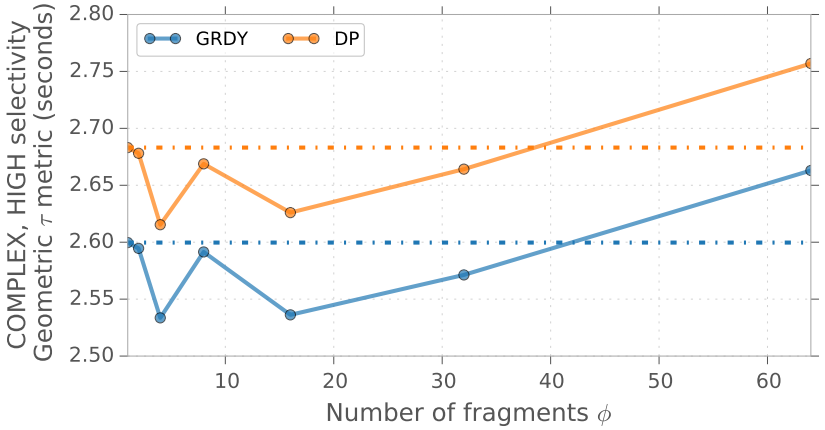


Fig. 20: High selectivity COMPLEX queries

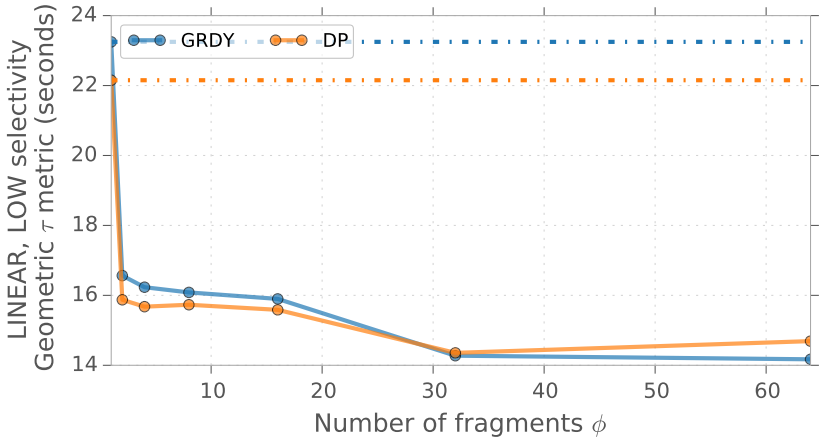


Fig. 21: Low selectivity LINEAR queries

range. Here the greedy optimiser (GRDY) seems to benefit the most by achieving the highest performance tradeoff for $\phi = 64$. In this case the total time stays relatively stable while the time for first results drops from ca. 7.5 seconds to ca. 4.75 seconds.

For COMPLEX queries the choice of number of fragments has a positive effect on the GRDY optimiser whose time for first results drops from ca. 155 seconds to ca. 40 seconds. The DP strategy appears to be less affected by fragmentation in this case. We attribute this to the fact that GRDY ends up choosing suboptimal plans where the effect of fragmentation is more dramatic, in contrast to DP which chooses optimal plans.

In conclusion, we can observe that in general the asynchronous GRDY and DP strategies, where each fragment is optimised either greedily or via dynamic programming in isolation and executed concurrently, do generally benefit up to a point from an increased number of fragments. The most impact can be observed for LINEAR and FLAKE low selectivity queries. We believe that this is the case due to the more flexible scheduling of

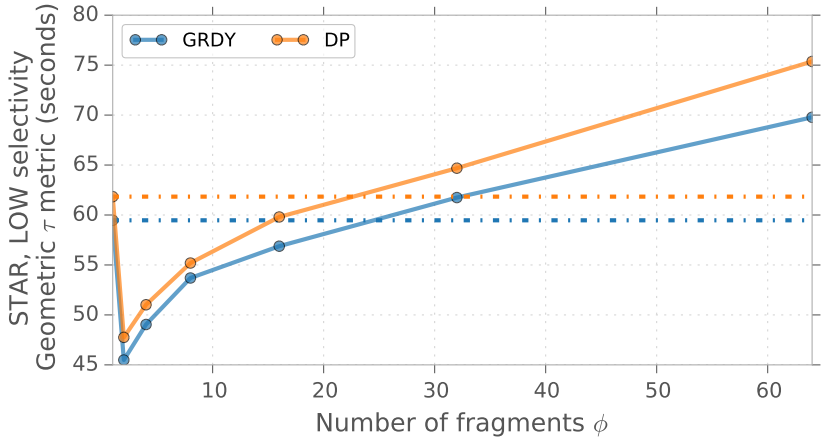


Fig. 22: Low selectivity STAR queries

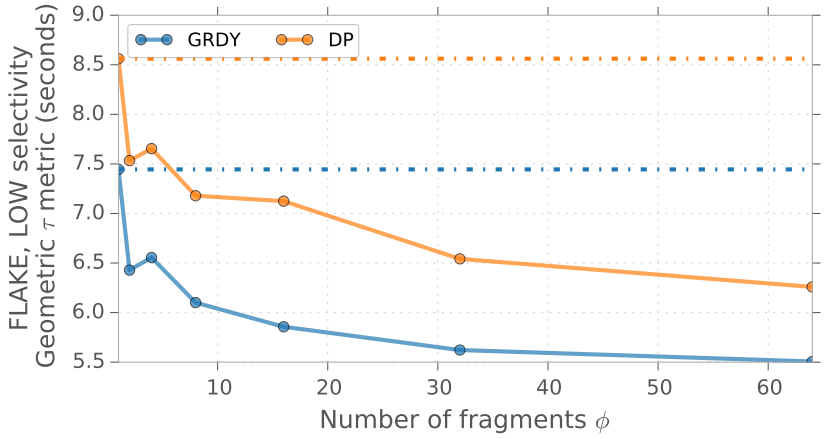


Fig. 23: Low selectivity FLAKE queries

resources, a direct consequence of the concurrent and asynchronous execution paradigm that X-AVALANCHE employs.

7 Limitations and Future Work

In the following we are going to detail X-AVALANCHE limitations as well as those of the system's optimisation methods and operator design. In addition, based on these limitations and findings, we will briefly discuss possible future work directions.

The work presented in this paper exhibits two kinds of limitations. First, X-AVALANCHE could be extended and/or optimised further and second, the external validity of our empirical evaluation is limited.

A first limitation stems from the fact that when employing *plan fragmentation* to derive an optimal tradeoff between total query execution time and time to first results

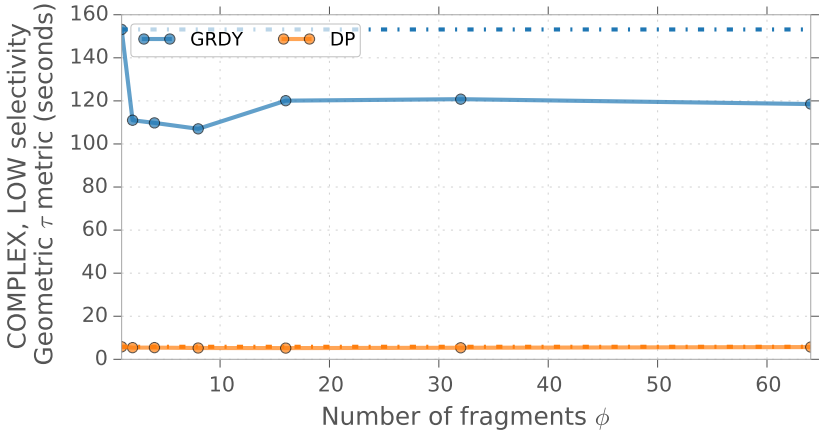


Fig. 24: Low selectivity COMPLEX queries

the choice between parametric and non-parametric space reduction algorithms is not automatic. Potential future work directions could include automatic learning of the parameters. This would entail learning which method to use and what is a good prior or number of fragments using of methods such as *Bayesian Optimisation* [Snoek et al., 2012] or self-tuning database methods [Chaudhuri and Narasayya, 2007].

Additional limitations stem from the *mismatch* between real and predicted plan performance. Traditional query optimisation algorithms like bottom up DP approaches assume that the cost model is optimal. In reality, plan cost estimations vary widely from their true cost. Consequently, fragmentation derived performance gains are diminished and depend on the estimative power of the cost model. Improving the cost model's accuracy will allow X-AVALANCHE to make better optimisation decisions and improve performance.

To further improve X-AVALANCHE's performance a number of research avenues and potential solutions stand out. While X-AVALANCHE extends and enhances the distributed state management protocol of AVALANCHE, it does not address all sources of limitation. One such limitation is derived from the level of impact that low performing SPARQL endpoints have on the system. While this is addressed to a certain degree by caching of result-sets, X-AVALANCHE does not cache SPARQL queries with VALUES bindings. A future extension could entail investigating how to use bloom filters [Broder and Mitzenmacher, 2003] to reduce the number of bindings sent to remote endpoints and therefore remote workload. Furthermore, X-AVALANCHE union operator is not optimised to take duplicates into account. On the Web of Data records are duplicated leading to a more optimisation possibilities by investigating the applicability of bloom filters to these cases or employing methods similar to the ones described in [Saleem et al., 2013].

Finally, the experimental setup relies on a limited number of physical resources. A physical machine is typically tasked with accommodating more than a dozen SPARQL and X-AVALANCHE endpoints. The resulting resource contention, generated by the competition for shared resources such as RAM, disk & network I/O, and CPU-time, can

have a negative impact on measured system performance, a fact that can be mitigated by the choice of physical machines.

8 Conclusions

To conclude, in this paper we present an extension of our original Avalanche SPARQL federation engine, which we call x-Avalanche. First, we introduce support for disjunctions when data is partitioned, by employing a novel parallel union algorithm called: *parallel tree union*. Results show that the parallel algorithm is able to perform up to **5x** faster than a naïve serial one. Second, we enhanced the distributed state management specific to our federated SPARQL protocol. To this end, each x-Avalanche operator is enhanced with support for execution by proxy allowing for orchestration effort offloading to other participating endpoints. In addition, we make use of parallel multicast bind-joins to minimise network traffic. At the same time we remotely cache query results (given allotted memory for cache) when no VALUES bindings are present in the query. This strategy alone, reduced overall query processing time by $\approx 10\%$.

Furthermore, we introduce a first novel approach to optimally reduce and traverse the *extended planning space*, that is suitable for large federations of RDF stores. We identify a new class of easily parallelizable plans we call *fragmented bushy plans* and we show how to optimally find the largest partial results set retrievable in the shortest possible time given external constraints. We implement and compare two exemplars of the non-parametric and parametric optimal extended planning space reduction methods: *bayesian-blocks* and *k-segmentation* and conclude in favour of the parametric approach, given its intrinsic control of the number of fragments. Finally, to support our hypothesis we also introduced a new synthetic benchmark designed with the difficult case of large homogenous RDF federations in mind. Released as open-source, the benchmark relies on LUBM to generate the data, which is than distributed to a given number of sites based on a user specified distribution. In addition, it borrows from the Waterloo SPARQL Diversity Test Suite (WatDiv) for query design.

Combined, x-Avalanche's enhancements and optimisations can lead to dramatic performance improvements over one of the top federated SPARQL engines to date: FedX – as shown in Saleem et al. [2014]. While, in the best case, x-Avalanche is up to **70x** times faster when delivering first results, it is also on average **more than 20 times faster for total query execution time** for low selectivity queries.

In summary, x-Avalanche shows that federated SPARQL processing can still be substantially improved both by focusing on low-level elements such as operator design and high-level system architecture considerations. As such we believe that the insight we gained from x-Avalanche provide an important building block for building the Web of Data.

Acknowledgements

This work was partially supported by the Swiss National Science Foundation under contract number 200021-118000.

Appendix

A Detailed Results and Statistics

The average time to retrieve the cardinality of a triple-pattern was **0.246** seconds with $\sigma = 0.01$ seconds.

B Benchmark Queries

B.1 Linear Queries

```
SELECT * WHERE {
?researchGroups lubm:subOrganizationOf ?department .
?department lubm:name "Department1" .}
```

Listing 6.1: LQ1

```
SELECT * WHERE {
?department lubm:subOrganizationOf ?university .
?professor lubm:worksFor ?department .
?student lubm:advisor ?professor .
?student lubm:memberOf <http://www.Department1.University0.edu> .}
```

Listing 6.2: LQ2

```
SELECT * WHERE {
?resgroup lubm:subOrganizationOf ?department .
?professor lubm:worksFor ?department .
?student lubm:advisor ?professor .
?student lubm:memberOf <http://www.Department1.University0.edu> .}
```

Listing 6.3: LQ3

```
SELECT * WHERE {
?advisor lubm:emailAddress ?email .
?advisor lubm:worksFor ?department .
?department lubm:name "Department1" .}
```

Listing 6.4: LQ4

B.2 Star Queries

```
SELECT * WHERE {
?student lubm:advisor ?advisor .
?student lubm:name ?name .
?student lubm:undergraduateDegreeFrom ?university .
```

```
?student lubm:takesCourse <http://www.Department1.University0.edu/GraduateCourse33
    > .}
```

Listing 6.5: LQ5

```
SELECT * WHERE {
?professor lubm:emailAddress ?mail .
?professor lubm:telephone ?phone .
?professor lubm:doctoralDegreeFrom ?doctor .
?professor lubm:name "FullProfessor1" .}
```

Listing 6.6: LQ6

```
SELECT * WHERE {
?student lubm:memberOf  ?department .
?student lubm:takesCourse  ?course .
?student lubm:advisor ?advisor .
?student lubm:teachingAssistantOf  ?tacourse .
?student lubm:emailAddress ?email .
?student lubm:name ?name .
?student lubm:telephone ?telephone .
?student lubm:undergraduateDegreeFrom <http://www.University0.edu> .}
```

Listing 6.7: LQ7

```
SELECT * WHERE {
?student lubm:memberOf  ?department .
?student lubm:takesCourse  ?course .
?student lubm:advisor ?advisor .
?student lubm:teachingAssistantOf  ?tacourse .
?student lubm:emailAddress ?email .
?student lubm:name "GraduateStudent71" .
?student lubm:telephone ?telephone .
?student lubm:undergraduateDegreeFrom ?university .}
```

Listing 6.8: LQ8

B.3 Snow Flake Queries

```
SELECT * WHERE {
?student lubm:advisor ?advisor .
?advisor lubm:worksFor ?department .
?department lubm:subOrganizationOf ?university .
?student lubm:name ?name .
?student lubm:telephone ?tel .
?student lubm:takesCourse <http://www.Department12.University1.edu/Course1> .}
```

Listing 6.9: LQ9

```
SELECT * WHERE {
?department lubm:name ?name .
?resgroup lubm:subOrganizationOf ?department .
```

```
?department lubm:subOrganizationOf <http://www.University0.edu> .
?student lubm:memberOf ?department .
?student lubm:advisor ?professor .
?student lubm:takesCourse ?course .}
```

Listing 6.10: LQ10

```
SELECT * WHERE {
?department lubm:name ?name .
?resgroup lubm:subOrganizationOf ?department .
?department lubm:subOrganizationOf ?university .
?student lubm:memberOf ?department .
?student lubm:advisor ?professor .
?student lubm:takesCourse <http://www.Department1.University0.edu/GraduateCourse33
    > .}
```

Listing 6.11: LQ11

B.4 Complex Queries

```
SELECT * WHERE {
?department lubm:subOrganizationOf ?university .
?resgroup lubm:subOrganizationOf ?department .
?student lubm:memberOf ?department .
?department lubm:name ?name .
?student lubm:advisor ?professor .
?publication lubm:publicationAuthor ?professor .
?publication lubm:publicationAuthor <http://www.Department1.University10.edu/
    AssociateProfessor1>.}
```

Listing 6.12: LQ12

```
SELECT * WHERE {
?department lubm:subOrganizationOf ?university .
?resgroup lubm:subOrganizationOf ?department .
?student lubm:memberOf ?department .
?student lubm:advisor ?professor .
?student lubm:takesCourse ?course .
?publication lubm:publicationAuthor ?professor .
?publication lubm:publicationAuthor <http://www.Department1.University10.edu/
    AssociateProfessor1> .
?publication lubm:name ?title .}
```

Listing 6.13: LQ13

```
SELECT * WHERE {
?student lubm:advisor ?advisor .
?advisor lubm:worksFor ?department .
?department lubm:subOrganizationOf <http://www.University0.edu> .
?head lubm:headOf ?department .
?head lubm:emailAddress ?email .
?head lubm:doctoralDegreeFrom ?alma .
?student lubm:name ?name .
```

```
?student lubm:telephone ?tel .
?student lubm:takesCourse ?course .}
```

Listing 6.14: LQ14

C Union Benchmark Queries

```
SELECT * WHERE{
?student lubm:takesCourse <http://www.Department12.University1.edu/Course1> }
```

Listing 6.15: LU1

```
SELECT * WHERE{
?department lubm:name "Department1" }
```

Listing 6.16: LU2

```
SELECT * WHERE{
?student  lubm:undergraduateDegreeFrom <http://www.University0.edu> }
```

Listing 6.17: LU3

```
SELECT * WHERE{
?professor lubm:name "FullProfessor1" }
```

Listing 6.18: LU4

References

Maribel Acosta, Maria-Esther Vidal, Tomas Lampo, Julio Castillo, and Edna Ruckhaus. ANAPSID: An adaptive query processing engine for SPARQL endpoints. In *The Semantic Web – ISWC 2011*, pages 18–34. Springer Science + Business Media, 2011. doi: 10.1007/978-3-642-25073-6_2. URL http://dx.doi.org/10.1007/978-3-642-25073-6_2.

K. Alexander, R. Cyganiak, M. Hausenblas, and J. Zhao. Describing Linked Datasets - On the Design and Usage of voiD, the 'Vocabulary of Interlinked Datasets'. In *WWW 2009 Workshop: Linked Data on the Web (LDOW2009)*, Madrid, Spain, 2009.

Marshall Van Alstyne, Erik Brynjolfsson, and Stuart Madnick. Why not one big database? principles for data ownership. *Decision Support Systems*, 15(4):267–284, dec 1995. doi: 10.1016/0167-9236(94)00042-4. URL http://dx.doi.org/10.1016/0167-9236(94)00042-4.

Güneş Aluç, Olaf Hartig, M. Tamer Özsu, and Khuzaima Daudjee. Diversified stress testing of RDF data management systems. In *The Semantic Web – ISWC 2014*, pages 197–212. Springer Science + Business Media, 2014. doi: 10.1007/978-3-319-11964-9_13. URL http://dx.doi.org/10.1007/978-3-319-11964-9_13.

Gennady Antoshenkov and Mohamed Ziauddin. Query processing and optimization in oracle rdb. *The VLDB Journal The International Journal on Very Large Data Bases*, 5(4):229–237, dec 1996. doi: 10.1007/s007780050026. URL http://dx.doi.org/10.1007/s007780050026.

M. M. Astrahan, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, V. Watson, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, and P. R. McJones. System r: relational approach to database management. *ACM Trans. Database Syst.*, 1(2):97–137, jun 1976. doi: 10.1145/320455.320457. URL http://dx.doi.org/10.1145/320455.320457.

Medha Atre, Vineet Chaoji, Mohammed J. Zaki, and James A. Hendler. Matrix "bit" loaded: A scalable lightweight join query processor for rdf data. In *Proceedings of the 19th international conference on World wide web - WWW '10*, pages 41–50. Association for Computing Machinery (ACM), 2010. doi: 10.1145/1772690.1772696. URL http://dx.doi.org/10.1145/1772690.1772696.

Cosmin Başca and Abraham Bernstein. Avalanche: putting the spirit of the web back into semantic web querying. In *Proceedings Of The 6th International Workshop On Scalable Semantic Web Knowledge Base Systems (SSWS2010)*, pages 64–79, November 2010. URL http://dx.doi.org/10.5167/uzh-44857.

Cosmin Başca and Abraham Bernstein. Querying a messy web of data with avalanche. *Web Semantics: Science, Services and Agents on the World Wide Web*, 26:1–28, may 2014. doi: 10.1016/j.websem.2014.04.002. URL http://dx.doi.org/10.1016/j.websem.2014.04.002.

R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, may 2001. URL http://www.sciam.com/article.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21.

Abraham Bernstein, Christoph Kiefer, and Markus Stocker. Optarq: A sparql optimization approach based on triple pattern selectivity estimation. Technical Report ifi-2007.03, University of Zürich, 2007.

Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data - the story so far. *International Journal on Semantic Web and Information Systems*, 5(3):1–22, 2009a. doi: 10.4018/jswis.2009081901. URL http://dx.doi.org/10.4018/jswis.2009081901.

Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data - the story so far. *International Journal on Semantic Web and Information Systems*, 5(3):1–22, 2009b. doi: 10.4018/jswis.2009081901. URL http://dx.doi.org/10.4018/jswis.2009081901.

Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2003. URL http://projecteuclid.org/euclid.im/1109191032.

Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF schema. In *The Semantic Web — ISWC 2002*, pages 54–68. Springer Science + Business Media, 2002. doi: 10.1007/3-540-48005-6_7. URL http://dx.doi.org/10.1007/3-540-48005-6_7.

Min Cai and Martin Frank. RDFPeers: A Scalable Distributed RDF Repository Based on a Structured Peer-to-peer Network. In *Proceedings of the 13th conference on World Wide Web - WWW '04*, pages 650–657. Association for Computing Machinery (ACM), 2004. doi: 10.1145/988672.988760. URL http://dx.doi.org/10.1145/988672.988760.

Surajit Chaudhuri and Vivek Narasayya. Self-tuning database systems: A decade of progress. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 3–14. VLDB Endowment, 2007. ISBN 978-1-59593-649-3. URL http://dl.acm.org/citation.cfm?id=1325851.1325856.

Richard Cyganiak. A relational algebra for SPARQL. Technical Report HPL-2005-170, HP Laboratories, Sep 2005. URL http://www.hpl.hp.com/techreports/2005/HPL-2005-170.pdf.

Jérôme David and Jérôme Euzenat. Comparison between ontology distances (preliminary results). In *The Semantic Web - ISWC 2008*, pages 245–260. Springer Science + Business Media, 2008. doi: 10.1007/978-3-540-88564-1_16. URL http://dx.doi.org/10.1007/978-3-540-88564-1_16.

Jérôme David, Jérôme Euzenat, and Ondřej Šváb-Zamazal. Ontology similarity in the alignment space. In *The Semantic Web – ISWC 2010*, pages 129–144. Springer Science + Business Media, 2010. doi: 10.1007/978-3-642-17746-0_9. URL http://dx.doi.org/10.1007/978-3-642-17746-0_9.

Amol Deshpande, Zachary Ives, and Vijayshankar Raman. Adaptive query processing. *Foundations and Trends® in*, 1(1):1–140, 2006. doi: 10.1561/1900000001. URL http://dx.doi.org/10.1561/1900000001.

Orri Erling and Ivan Mikhailov. RDF support in the virtuoso DBMS. In *Networked Knowledge - Networked Media*, pages 7–24. Springer Science + Business Media, 2009.

doi: 10.1007/978-3-642-02184-8_2. URL http://dx.doi.org/10.1007/978-3-642-02184-8_2.

C. Estan and J.F. Naughton. End-biased samples for join cardinality estimation. In *22nd International Conference on Data Engineering (ICDE'06)*. Institute of Electrical & Electronics Engineers (IEEE), 2006. doi: 10.1109/icde.2006.61. URL http://dx.doi.org/10.1109/ICDE.2006.61.

Jerome Euzenat and Pavel Shvaiko. *Ontology Matching*. Springer Science + Business Media, 2007. doi: 10.1007/978-3-540-49612-0. URL http://dx.doi.org/10.1007/978-3-540-49612-0.

Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology*, 2(2):115–150, may 2002. doi: 10.1145/514183.514185. URL http://dx.doi.org/10.1145/514183.514185.

Atemezing Ghislain, Corcho Óscar, Garijo Daniel, Mora José, Poveda-Villalón María, Rozas Pablo, Vila-Suero Daniel, and Villazón-Terrazas Boris. Transforming meteorological data into linked data. *Semantic Web*, 4(3):285–290, 2013. ISSN 1570-0844. doi: 10.3233/SW-120089. URL http://doi.org/10.3233/SW-120089.

Olaf Görlitz and Steffen Staab. SPLENDID: SPARQL endpoint federation exploiting VOID descriptions. In *Proceedings of the Second International Workshop on Consuming Linked Data (COLD2011), Bonn, Germany, October 23, 2011*, 2011. URL http://ceur-ws.org/Vol-782/GoerlitzAndStaab_COLD2011.pdf.

Christophe Guéret, Eyal Oren, Stefan Schlobach, and Martijn Schut. An evolutionary perspective on approximate RDF query answering. In *Scalable Uncertainty Management*, pages 215–228. Springer Science + Business Media, 2008. doi: 10.1007/978-3-540-87993-0_18. URL http://dx.doi.org/10.1007/978-3-540-87993-0_18.

Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):158–182, oct 2005. doi: 10.1016/j.websem.2005.06.005. URL http://dx.doi.org/10.1016/j.websem.2005.06.005.

Harry Halpin, Patrick J. Hayes, James P. McCusker, Deborah L. McGuinness, and Henry S. Thompson. When owl:sameAs isn't the same: An analysis of identity in linked data. In *The Semantic Web – ISWC 2010*, pages 305–320. Springer Science + Business Media, 2010. doi: 10.1007/978-3-642-17746-0_20. URL http://dx.doi.org/10.1007/978-3-642-17746-0_20.

Andreas Harth, Jürgen Umbrich, Aidan Hogan, and Stefan Decker. YARS2: A federated repository for querying graph structured data from the web. In *The Semantic Web*, pages 211–224. Springer Science + Business Media, 2007. doi: 10.1007/978-3-540-76298-0_16. URL http://dx.doi.org/10.1007/978-3-540-76298-0_16.

Andreas Harth, Katja Hose, Marcel Karnstedt, Axel Polleres, Kai-Uwe Sattler, and Jürgen Umbrich. Data summaries for on-demand queries over linked data. In *Proceedings of the 19th international conference on World wide web - WWW '10*. Association for Computing Machinery (ACM), 2010. doi: 10.1145/1772690.1772733. URL http://dx.doi.org/10.1145/1772690.1772733.

Olaf Hartig. SQUIN. In *Proceedings of the 2013 international conference on Management of data - SIGMOD '13*. Association for Computing Machinery (ACM), 2013. doi: 10.1145/2463676.2465231. URL http://dx.doi.org/10.1145/2463676.2465231.

Olaf Hartig and Ralf Heese. The SPARQL query graph model for query optimization. In *The Semantic Web: Research and Applications*, pages 564–578. Springer Science + Business Media, 2007. doi: 10.1007/978-3-540-72667-8_40. URL http://dx.doi.org/10.1007/978-3-540-72667-8_40.

Olaf Hartig, Christian Bizer, and Johann-Christoph Freytag. Executing SPARQL queries over the web of linked data. In *The Semantic Web - ISWC 2009*, pages 293–309. Springer Science + Business Media, 2009a. doi: 10.1007/978-3-642-04930-9_19. URL http://dx.doi.org/10.1007/978-3-642-04930-9_19.

Olaf Hartig, Christian Bizer, and Johann-Christoph Freytag. Executing SPARQL queries over the web of linked data. In *The Semantic Web - ISWC 2009*, pages 293–309. Springer Science + Business Media, 2009b. doi: 10.1007/978-3-642-04930-9_19. URL http://dx.doi.org/10.1007/978-3-642-04930-9_19.

Dennis Heimbigner and Dennis McLeod. A federated architecture for information management. *TOIS*, 3(3):253–278, jul 1985. doi: 10.1145/4229.4233. URL http://dx.doi.org/10.1145/4229.4233.

Herodotos Herodotou, Nedyalko Borisov, and Shivnath Babu. Query optimization techniques for partitioned tables. In *Proceedings of the 2011 international conference on Management of data - SIGMOD '11*. Association for Computing Machinery (ACM), 2011. doi: 10.1145/1989323.1989330. URL http://dx.doi.org/10.1145/1989323.1989330.

Katja Hose and Ralf Schenkel. Towards benefit-based RDF source selection for SPARQL queries. In *Proceedings of the 4th International Workshop on Semantic Web Information Management - SWIM '12*. Association for Computing Machinery (ACM), 2012. doi: 10.1145/2237867.2237869. URL http://dx.doi.org/10.1145/2237867.2237869.

Jiewen Huang, Daniel J Abadi, and Kun Ren. Scalable sparql querying of large rdf graphs. *Proceedings of the VLDB Endowment*, 4(11):1123–1134, 2011.

Yannis E. Ioannidis and Stavros Christodoulakis. On the propagation of errors in the size of join results. *ACM SIGMOD Record*, 20(2):268–277, apr 1991. doi: 10.1145/119995.115835. URL http://dx.doi.org/10.1145/119995.115835.

Borislav Iordanov. HyperGraphDB: A generalized graph database. In *Web-Age Information Management*, pages 25–36. Springer Science + Business Media, 2010. doi: 10.1007/978-3-642-16720-1_3. URL http://dx.doi.org/10.1007/978-3-642-16720-1_3.

Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997. ISBN 0-201-89683-4.

Donald Kossmann. The state of the art in distributed query processing. *CSUR*, 32 (4):422–469, dec 2000. doi: 10.1145/371578.371598. URL http://dx.doi.org/10.1145/371578.371598.

Donald Kossmann and Konrad Stocker. Iterative dynamic programming: a new class of query optimization algorithms. *ACM Trans. Database Syst.*, 25(1):43–82, mar 2000. doi: 10.1145/352958.352982. URL http://dx.doi.org/10.1145/352958.352982.

Günter Ladwig and Thanh Tran. SIHJoin: Querying remote and local linked data. In *The Semantic Web: Research and Applications*, pages 139–153. Springer Science + Business Media, 2011. doi: 10.1007/978-3-642-21034-1_10. URL http://dx.doi.org/10.1007/978-3-642-21034-1_10.

Andreas Langegger and Wolfram Woss. RDFStats - an extensible RDF statistics generator and library. In *2009 20th International Workshop on Database and Expert Systems Application*. Institute of Electrical & Electronics Engineers (IEEE), 2009. doi: 10.1109/dexa.2009.25. URL http://dx.doi.org/10.1109/DEXA.2009.25.

Andreas Langegger, Wolfram Wöß, and Martin Blöchl. A semantic web middleware for virtual data integration on the web. In *The Semantic Web: Research and Applications*, pages 493–507. Springer Science + Business Media, 2008. doi: 10.1007/978-3-540-68234-9_37. URL http://dx.doi.org/10.1007/978-3-540-68234-9_37.

Yingjie Li and Jeff Heflin. Using reformulation trees to optimize queries over distributed heterogeneous sources. In *The Semantic Web – ISWC 2010*, pages 502–517. Springer Science + Business Media, 2010. doi: 10.1007/978-3-642-17746-0_32. URL http://dx.doi.org/10.1007/978-3-642-17746-0_32.

Steven Lynden, Isao Kojima, Akiyoshi Matono, and Yusuke Tanimura. ADERIS: An adaptive query processor for joining federated SPARQL endpoints. In *Lecture Notes in Computer Science*, pages 808–817. Springer Science + Business Media, 2011. doi: 10.1007/978-3-642-25106-1_28. URL http://dx.doi.org/10.1007/978-3-642-25106-1_28.

Angela Maduko, Kemafor Anyanwu, Amit Sheth, and Paul Schliekelman. Estimating the cardinality of RDF graph patterns. In *Proceedings of the 16th international conference on World Wide Web - WWW '07*. Association for Computing Machinery (ACM), 2007. doi: 10.1145/1242572.1242782. URL http://dx.doi.org/10.1145/1242572.1242782.

Norbert Martínez-Bazan, Victor Muntés-Mulero, Sergio Gómez-Villamor, Jordi Nin, Mario-A. Sánchez-Martínez, and Josep-L. Larriba-Pey. Dex. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management - CIKM '07*. Association for Computing Machinery (ACM), 2007. doi: 10.1145/1321440.1321521. URL http://dx.doi.org/10.1145/1321440.1321521.

Kaisa Miettinen. *Nonlinear Multiobjective Optimization*. Springer US, 1998. doi: 10.1007/978-1-4615-5563-6. URL http://dx.doi.org/10.1007/978-1-4615-5563-6.

Hannes Muhleisen, Tilman Walther, Anne Augustin, Marko Harasic, and Robert Tolksdorf. Configuring a self-organized semantic storage service. In *Proceedings Of The 6th International Workshop On Scalable Semantic Web Knowledge Base Systems (SSWS2010)*, pages 1–16, November 2010.

Wolfgang Nejdl, Boris Wolf, Changtao Qu, Stefan Decker, Michael Sintek, Ambjörn Naeve, Mikael Nilsson, Matthias Palmér, and Tore Risch. EDUTELLA: a p2p networking infrastructure based on rdf. In *Proceedings of the eleventh international conference on World Wide Web - WWW '02*, pages 604–615. Association for Computing

Machinery (ACM), 2002. doi: 10.1145/511446.511525. URL http://dx.doi.org/10.1145/511446.511525.

Thomas Neumann and Guido Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *2011 IEEE 27th International Conference on Data Engineering*. Institute of Electrical & Electronics Engineers (IEEE), apr 2011. doi: 10.1109/icde.2011.5767868. URL http://dx.doi.org/10.1109/ICDE.2011.5767868.

Thomas Neumann and Gerhard Weikum. Scalable join processing on very large RDF graphs. In *Proceedings of the 35th SIGMOD international conference on Management of data - SIGMOD '09*. Association for Computing Machinery (ACM), 2009a. doi: 10.1145/1559845.1559911. URL http://dx.doi.org/10.1145/1559845.1559911.

Thomas Neumann and Gerhard Weikum. The RDF-3x engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113, sep 2009b. doi: 10.1007/s00778-009-0165-y. URL http://dx.doi.org/10.1007/s00778-009-0165-y.

Thomas Neumann and Gerhard Weikum. x-RDF-3x. *Proceedings of the VLDB Endowment*, 3(1-2):256–263, sep 2010. doi: 10.14778/1920841.1920877. URL http://dx.doi.org/10.14778/1920841.1920877.

Andriy Nikolov, Andreas Schwarte, and Christian Hütter. FedSearch: Efficiently combining structured queries and full-text search in a SPARQL federation. In *The Semantic Web – ISWC 2013*, pages 427–443. Springer Science + Business Media, 2013. doi: 10.1007/978-3-642-41335-3_27. URL http://dx.doi.org/10.1007/978-3-642-41335-3_27.

Kiyoshi Ono and Guy M. Lohman. Measuring the complexity of join enumeration in query optimization. In *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 314–325. Morgan Kaufmann, 1990. ISBN 1-55860-149-X.

Eyal Oren, Christophe Guéret, and Stefan Schlobach. Anytime query answering in RDF through evolutionary algorithms. In *The Semantic Web - ISWC 2008*, pages 98–113. Springer Science + Business Media, 2008. doi: 10.1007/978-3-540-88564-1_7. URL http://dx.doi.org/10.1007/978-3-540-88564-1_7.

M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems (2Nd Ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999. ISBN 0-13-659707-6.

E. S. PAGE. A test for a change in a parameter occurring at an unknown point. *Biometrika*, 42(3-4):523–527, 1955. doi: 10.1093/biomet/42.3-4.523. URL http://dx.doi.org/10.1093/biomet/42.3-4.523.

Bastian Quilitz and Ulf Leser. Querying distributed RDF data sources with SPARQL. In *The Semantic Web: Research and Applications*, pages 524–538. Springer Science + Business Media, 2008. doi: 10.1007/978-3-540-68234-9_39. URL http://dx.doi.org/10.1007/978-3-540-68234-9_39.

Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, 3 edition, 2003. ISBN 0072465638, 9780072465631.

Sukriti Ramesh, Odysseas Papapetrou, and Wolf Siberski. Optimizing distributed joins with bloom filters. In *Distributed Computing and Internet Technology*, pages 145–156.

Springer Science + Business Media, 2009. doi: 10.1007/978-3-540-89737-8_15. URL http://dx.doi.org/10.1007/978-3-540-89737-8_15.

Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases*. O'Reilly Media, Inc., 2013. ISBN 1449356265, 9781449356262.

Muhammad Saleem, Axel-Cyrille Ngonga Ngomo, Josiane Xavier Parreira, Helena F. Deus, and Manfred Hauswirth. DAW: Duplicate-AWare federated query processing over the web of data. In *The Semantic Web – ISWC 2013*, pages 574–590. Springer Science + Business Media, 2013. doi: 10.1007/978-3-642-41335-3_36. URL http://dx.doi.org/10.1007/978-3-642-41335-3_36.

Muhammad Saleem, Yasar Khan, Ali Hasnain, Ivan Ermilov, and Axel-Cyrille Ngonga Ngomo. A fine-grained evaluation of SPARQL endpoint federation systems. *Semantic Web Journal*, 2014.

Jeffrey D. Scargle, Jay P. Norris, Brad Jackson, and James Chiang. STUDIES IN ASTRONOMICAL TIME SERIES ANALYSIS. VI. BAYESIAN BLOCK REPRESENTATIONS. *ApJ*, 764(2):167, feb 2013. doi: 10.1088/0004-637x/764/2/167. URL http://dx.doi.org/10.1088/0004-637X/764/2/167.

Simon Schenk and Steffen Staab. Networked graphs. In *Proceeding of the 17th international conference on World Wide Web - WWW '08*. Association for Computing Machinery (ACM), 2008. doi: 10.1145/1367497.1367577. URL http://dx.doi.org/10.1145/1367497.1367577.

Max Schmachtenberg, Christian Bizer, and Heiko Paulheim. Adoption of the linked data best practices in different topical domains. In *The Semantic Web – ISWC 2014*, pages 245–260. Springer Science + Business Media, 2014. doi: 10.1007/978-3-319-11964-9_16. URL http://dx.doi.org/10.1007/978-3-319-11964-9_16.

Michael Schmidt, Olaf Görlitz, Peter Haase, Günter Ladwig, Andreas Schwarte, and Thanh Tran. FedBench: A benchmark suite for federated semantic data query processing. In *The Semantic Web – ISWC 2011*, pages 585–600. Springer Science + Business Media, 2011. doi: 10.1007/978-3-642-25073-6_37. URL http://dx.doi.org/10.1007/978-3-642-25073-6_37.

Andreas Schwarte, Peter Haase, Katja Hose, Ralf Schenkel, and Michael Schmidt. FedX: Optimization techniques for federated query processing on linked data. In *The Semantic Web – ISWC 2011*, pages 601–616. Springer Science + Business Media, 2011. doi: 10.1007/978-3-642-25073-6_38. URL http://dx.doi.org/10.1007/978-3-642-25073-6_38.

N. Shadbolt, T. Berners-Lee, and W. Hall. The semantic web revisited. *IEEE Intelligent Systems*, 21(3):96–101, may 2006. doi: 10.1109/mis.2006.62. URL http://dx.doi.org/10.1109/MIS.2006.62.

Amit P. Sheth and James A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *CSUR*, 22(3):183–236, sep 1990. doi: 10.1145/96602.96604. URL http://dx.doi.org/10.1145/96602.96604.

Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems 25*, pages 2951–2959. Curran Associates, Inc., 2012. URL http://papers.nips.cc/paper/4522-practical-bayesian-optimization-of-machine-learning-algorithms.pdf.

Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *Proceeding of the 17th international conference on World Wide Web - WWW '08*. Association for Computing Machinery (ACM), 2008a. doi: 10.1145/1367497.1367578. URL http://dx.doi.org/10.1145/1367497.1367578.

Markus Stocker, Andy Seaborne, Abraham Bernstein, Christoph Kiefer, and Dave Reynolds. SPARQL basic graph pattern optimization using selectivity estimation. In *Proceeding of the 17th international conference on World Wide Web - WWW '08*. Association for Computing Machinery (ACM), 2008b. doi: 10.1145/1367497.1367578. URL http://dx.doi.org/10.1145/1367497.1367578.

Heiner Stuckenschmidt, Richard Vdovjak, Geert-Jan Houben, and Jeen Broekstra. Index structures and algorithms for querying distributed RDF repositories. In *Proceedings of the 13th conference on World Wide Web - WWW '04*. Association for Computing Machinery (ACM), 2004. doi: 10.1145/988672.988758. URL http://dx.doi.org/10.1145/988672.988758.

Petros Tsialiamanis, Lefteris Sidirourgos, Irini Fundulaki, Vassilis Christophides, and Peter Boncz. Heuristics-based query optimisation for SPARQL. In *Proceedings of the 15th International Conference on Extending Database Technology - EDBT '12*. Association for Computing Machinery (ACM), 2012. doi: 10.1145/2247596.2247635. URL http://dx.doi.org/10.1145/2247596.2247635.

Giovanni Tummarello, Richard Cyganiak, Michele Catasta, Szymon Danielczyk, Renaud Delbru, and Stefan Decker. Sig.ma: Live views on the web of data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 8(4):355–364, nov 2010. doi: 10.1016/j.websem.2010.08.003. URL http://dx.doi.org/10.1016/j.websem.2010.08.003.

Kostas Tzoumas, Amol Deshpande, and Christian S. Jensen. Efficiently adapting graphical models for selectivity estimation. *The VLDB Journal*, 22(1):3–27, nov 2012. doi: 10.1007/s00778-012-0293-7. URL http://dx.doi.org/10.1007/s00778-012-0293-7.

Octavian Udrea, Andrea Pugliese, and V. S. Subrahmanian. Grin: A graph based rdf index. In *Proceedings of the 22Nd National Conference on Artificial Intelligence - Volume 2*, AAAI'07, pages 1465–1470. AAAI Press, 2007. ISBN 978-1-57735-323-2. URL http://dl.acm.org/citation.cfm?id=1619797.1619880.

J. Umbrich, C. Gutierrez, A. Hogan, M. Karnstedt, and Josiane Xavier Parreira. Eight fallacies when querying the web of data. In *2013 IEEE 29th International Conference on Data Engineering Workshops (ICDEW)*. Institute of Electrical & Electronics Engineers (IEEE), apr 2013. doi: 10.1109/icdew.2013.6547418. URL http://dx.doi.org/10.1109/ICDEW.2013.6547418.

Xin Wang, Thanassis Tiropanis, and Hugh C. Davis. LHD: optimising linked data query processing using parallelisation. In *Proceedings of the WWW2013 Workshop on Linked Data on the Web, Rio de Janeiro, Brazil, 14 May, 2013*, 2013. URL http://ceur-ws.org/Vol-996/papers/ldow2013-paper-06.pdf.

Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore. *Proceedings of the VLDB Endowment*, 1(1):1008–1019, aug 2008. doi: 10.14778/1453856.1453965. URL http://dx.doi.org/10.14778/1453856.1453965.

M. Zeleny. Compromise programming. In *Multiple Criteria Decision Making*, pages 262–301. University of South Carolina Press, Columbia, 1973.

Jan Zemanek, Simon Schenk, and Vojtech Svatek. Optimizing sparql queries over disparate rdf data sources through distributed semi-joins. In *ISWC 2008 Poster and Demo Session Proceedings*. CEUR-WS, 2008.

List of Figures